Intel® Technology Journal

Network Processors

This issue of Intel Technology Journal (Volume 6, Issue 3, 2002) explores the exciting advances in the development of network processors and shows how they are fast becoming the silicon core of next-generation networking.

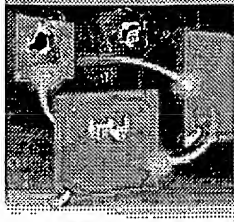### Inside you'll find the following papers:

The Next Generation of Intel IXP Network Processors

IXA Portability Framework: Preserving Software Investment in Network Processor Applications

Network Processor Performance Analysis Methodology

Network Processor Building Blocks for All-IP Wireless Networks

Packet over SONET: Achieving 10 Gigabit/sec. Packet Processing with an IXP2800

Implementing Voice over AAL2 on a Network Processor

Security: Adding Protection to the Network via the Network Processor

Challenges and Methodologies for Implementing High-Performance Network Processors

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Intel® Technology Journal

Network Processors

Articles

Preface Q3, 2002 Intel Technology Journal
By Lin Chao

Network processors are microprocessors designed specifically for packet processing, a core element of high-speed communication routers. Despite the recent communications industry downturn, network processor technologies continue to advance rapidly. Similar to the microprocessors used in a PC, a network processor is a programmable chip, but its instruction set has been optimized to support the operations that are needed in networking, especially for packet processing. A number of companies, including Intel Corporation, have developed network processors. The Intel network processors are based on the Intel® Internet Exchange Architecture (Intel® IXA), and include the IXP2800, IXP2400, and IXP1200 processors.

The trend toward using network processors is exciting for two main reasons. First, the use of network processors provides a way of developing high-speed routers, using standard, off-the-shelf hardware. Until now, the main router functions have been performed either in software (for the low-end routers) or with the use of expensive custom hardware (for high-end routers). Second, because the network processors are programmable, they support the development of high-speed routers that are flexible, in the sense that their functionality can be easily modified or extended by downloading new code. A network vendor could develop this code, or the code could potentially come from a third party or from the users of the equipment. This means that network processors would be the foundation for programmable or active networks.

There are three important elements in Intel's IXA as a network processing architecture.

Microengine technology – a subsystem of programmable, multi-threaded microengines that has high-performance packet processing.

Intel® XScale™ technology – provides the highest performance-to-power ratio in the industry, with performance up to 1000 MIPS and power consumption as low as 10mW.

The Intel IXA Portability Framework – a modular programming framework providing software code portability advantages.

This issue (Vol. 6 Issue 3, 2002) of Intel Technology Journal gives a detailed look into the exciting advances in the development of network processors. The first three papers explain Intel's next generation of network processor architecture, network security, and chip design challenges. The fourth and fifth papers look at performance methodologies and how to achieve high-rate packet throughput. The sixth paper discusses the IXA Portability Framework, which provides developers with software portability. The last two papers address voice over AAL2 on a network processor and building wireless networks. These eight papers show how network processors are fast becoming the silicon core of next-generation networking.

Innovations in Network Processors
By Jim Finnegan
Co-General Manager, Network Processor Division, Intel Communications Group

Moore's Law has a timeless reassuring certainty, yielding a relentless improvement in the cost and performance of the PC/workstation/server. While there is an obvious correlation between the speed of networks required to effectively distribute such PCs/workstations/servers, a less intuitive requirement for networks is the ability to simultaneously support different traffic types (voice, data, video), each with its own disparate properties of latency, error-rate tolerance, jitter, etc. Even though competing networking technologies, such as ATM, frame relay, and SNA, have converged on TCP/IP as the ubiquitous networking protocol, many generic challenges remain, such as address space (IPV6) and security. Furthermore, customer innovation demands the ability to inspect each packet payload and even routing decisions based upon content!

The specialized packet-processing properties of networking have requirements beyond those of general-purpose microprocessors. Historically, networking equipment manufacturers have been forced to develop ASICs for this task. There are a number of disadvantages with ASICs; notably, they are inherently inflexible and expensive to develop. They also trail emerging standards and invariably are implemented in a lagging process technology. Such a scenario has presented Intel with the huge opportunity to develop a new class of processor–the *network processor*. This issue of *Intel Technology Journal* is dedicated to network processors.

As stated above, the genesis of network processors came from the realization that an ASIC-based design approach offers neither the flexibility nor the time-to-market benefits required for increasingly higher data rates, evolving networking standards, and customer demand for extensive and versatile packet/cell processing capability. The classical networking challenge is to maintain stability while maximizing throughput and minimizing latency for the worst-case traffic. The case of OC-192 (10 Gigabits/sec) Packet over SONET (POS) for minimum packet size (49 bytes) presents significant processing and memory bandwidth challenges, equating to a throughput of 28 million packets per second or service time of 4.57 usecs per packet for transmit and receive. Considering the array of tasks per packet (e.g., route lookup, metering, policing, congestion avoidance, statistics, transmit scheduling, etc.), it is necessary to be able to sustain a processing capability of over 23,000 MIPs and packet memory bandwidth in excess of 40 Gigabits/sec. The introduction of Intel's latest members of the IXP network processor family solves this difficult problem using the IXP2800, as well as using the IXP2400 to solve similar problems at lower line rates.

One of the distinguishing features of the IXP network processors is the microengine cluster, which provides the demanding processing capability to service the packet arrival rate. There are 16 microengines running at 1.4GHz in the IXP2800, while the IXP2400 can service OC-48 traffic with 8 microengines running at up to 600MHz. Each microengine has a six-stage pipeline, low-latency local memory, and support of multiple threads to maximize utilization of processing capabilities. Since OC-192 packets are processed in real time, an elegant architectural feature was added to the IXP2800: a

full-duplex cryptographic unit. This was added to support multiple encryption standards—Data Encryption Standard (DES), Advanced Encryption Standard (AES), Secure Hash Algorithm (SHA)—providing on-chip performance to encrypt and authenticate Ipsec at 10 Gigabits/sec when all of the traffic needs to be secured.

A number of detailed performance analyses for OC-192 POS and OC-48 voice over AAL2 illustrate how to allocate the tasks per microengine in order to meet the very challenging performance requirements. What is probably evident is the need to present the users of these powerful network processors with a rich array of tools to develop their respective applications. This has been addressed by providing a set of well-instrumented development tools, pivotal to which is the Microengine C Compiler, deemed crucial to our customers' goals for software reuse. A formal software framework enables the customer to accelerate software development and maximize reuse.

Reuse, in fact, is a key principle used in the development of these high-performance processors, by which RTL is shared across geographically dispersed teams with different target frequencies and process technologies. An additional item of note in the development methodology is the innovative internally developed VMOD tool that generates RTL and the corresponding C++ cycle-accurate model (*Transactor*), which interfaces seamlessly with the development environment (*Workbench*).

The introduction of these new products to the IXP family confirms Intel's technology, performance, and innovation leadership position in a new and exciting network processor market.

The Next Generation of Intel IXP Network Processors

Matthew Adiletta, Mark Rosenbluth, Debra Bernstein,
Gilbert Wolrich, Hugh Wilkinson
Intel Communications Group, Intel Corporation

Index words: network processors, IXP, communication architecture, routing, switching, Ethernet, ATM, multi-service switches, multi-processors, microprocessor architecture, multi-threading, 10Gb/s, OC-192, OC-48.

ABSTRACT

This paper describes the next generation of Intel Internet eXchange Processors (IXPs). The IXP family of network processors is growing with the addition of three new parts. This paper focuses on the high-end IXP2800. The IXP2800 is capable of 10Gb/s ATM, OC-192 POS, or Ethernet data processing. The IXP2400 is a sibling and is capable of sustained OC-48 or Quad Gigabit Ethernet data processing. The third new member of the family is the IXP440, which is a Customer Premise Equipment (CPE) class device. The IXP2800 and IXP2400 share the architectural chassis, major functional units and software programming model, as well as the same instruction set.

This paper covers system architecture, micro-architecture, and functional unit characteristics, and provides insights into the challenges of processing an incoming cell or packet every 35ns. Special attention is paid to the problem of enqueuing and dequeuing onto and from a linked list that is maintained in external memory. The challenge is that cell and packet arrival rates are approaching the external memory access latencies.

Finally, this paper concludes with future directions for the IXP family.

INTRODUCTION

The IXP1200 is the first member of the IXP network processor family. It has been designed into over 200 products at a wide range of companies and market segments. Introduced in 1999, it provided OC-12 or Gigabit Ethernet packet processing capability. The IXP2800 and IXP2400 leverage many learnings from the experiences of the IXP1200. In particular, refining the computational needs and memory access bandwidths required at different incoming line rates has led to providing both greater computational capability and memory bandwidth. The IXP2800 provides over 23,000 MIPs, the IXP2400 4800 MIPs, and the IXP1200 1200 MIPs.

There are many competing approaches to network processing. One approach is through dedicated hardware state machines with configurability, or minimal software programming capability. Another approach is through very high-performance microprocessors that are provided with a very flexible software programming capability. The IXP family employs the flexible software approach, with state-of-the-art compilers and debuggers. This allows the IXP to address many market segments and allows our customers to develop a base hardware platform that they can then use in different applications. Additionally, by providing a flexible software platform, customers can download features and capabilities to enhance product lifespans and product experiences.

The first section of this paper describes three system architectures using the IXP2800. It is interesting to note that the system architectures detailed may also be applied to the IXP2400, albeit at a lower incoming cell or packet rate.

The second section focuses on the internal architecture of the IXP2800. The chassis, or the interconnection between the different functional units, will be described, as well as the major functional units.

The third section provides details on the microengine, which is the processor arrayed in either 16 instantiations for the IXP2800 or eight instantiations for the IXP2400.

The challenges of packet or cell processing at 10 gigabits per second are then described along with the solution employed by the IXP2800. Then flexibility versus software complexity is discussed. The paper concludes with a discussion of the future directions of the IXP high-end processor.

IXP2800 SYSTEM EXAMPLES

Many system architectures are possible employing the IXP2800. This section details three configurations of IXP2800 processors supporting various switching applications.

Metro-LAN 10 Gigabit Ethernet Switching or OC-192 Packet over SONET Switching Blade

In this system architecture, two IXP2800 network processors are used (Figure 1). The top IXP2800 is used for ingress processing. Ingress processing tasks may include classification, metering, policing, congestion avoidance, statistics, segmentation, and traffic scheduling into a switching fabric. The bottom IXP2800 is used for egress processing. Egress processing tasks may similarly include reassembly, congestion avoidance, statistics, and traffic shaping. Both input and output buffering are supported using small DRAM buffers linked together by linked lists maintained in SRAM.
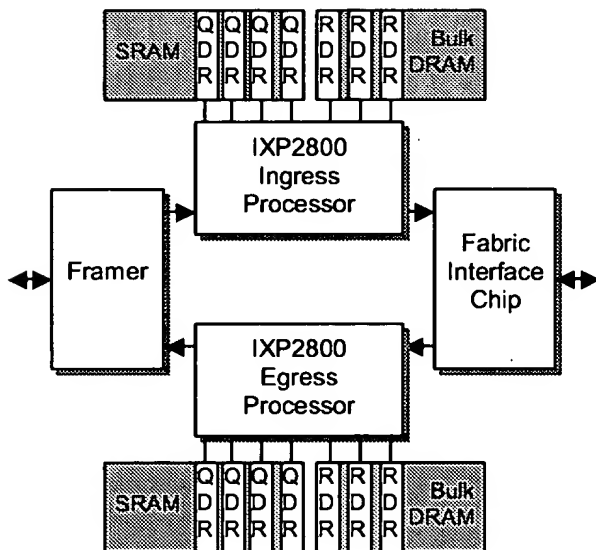

Figure 1: Metro-LAN 10 Gigabit Ethernet switching or OC-192 Packet over SONET configuration

The framer interfaces to the two IXP2800 network processors using the interface defined by the Optical Internetworking Forum SPI-4.2 Implementation Agreement. The fabric interfaces to the two IXP2800 network processors using the Common Switch Interface Specification-L1 (CSIX-L1) protocol implemented on top of the SPI-4.2 physical signaling. The ingress and egress network processors present a single full-duplex interface to the fabric, as if they were a single chip.

Packets are streamed into the ingress IXP2800 at or above line rate. The processing of a packet begins upon receipt of the initial part. The parts of a packet are received, reassembled, processed, buffered into DRAM, and enqueued for transmission into the fabric. Subsequently, the packet is scheduled and transmitted into the fabric to be processed by an egress IXP2800. The egress IXP2800 reassembles the packet in DRAM and queues the packet for outgoing transmission. Subsequently, the packet is transmitted out the egress framer. At both the ingress IXP2800 and the egress IXP2800, packet data is written to and read from DRAM only a single time. The DRAM interface consists of three Rambus DRAM (RDRAM) channels operating at a clock rate of up to 533MHz, offering an aggregate peak bandwidth of 51Gb/s.

At a maximum packet rate of approximately 15 million packets per second for 10 Gigabit Ethernet, the IXP2800 supports a service time of 8.53 usec per packet for receive and transmit processing by distributing the processing across 128 different computation threads. The IXP2800 can support the execution of up to 1493 microengine instructions per packet (93 instructions per microengine * 16 microengines) at this packet rate and a clock rate of 1.4GHz.

At a maximum packet rate of approximately 28 million packets per second for OC-192 Packet over SONET, the IXP2800 supports a service time of 4.57 usec per packet for receive and transmit. The IXP2800 can support the execution of up to 800 microengine instructions per packet at this packet rate and a clock rate of 1.4GHz.

The IXP2800 supports four QDR II SRAM interfaces that may be clocked at up to 250MHz. Each interface supports an independent read and write port, providing an aggregate read rate of 32Gb/s and, simultaneously, an aggregate write rate of 32Gb/s. These interfaces may be used to access SRAM. Additionally, Ternary Content Adressable Memories (TCAM), which support the same interface, are becoming available.

Classification may be performed using TRIE data structures in SRAM, hashing and collision resolution in SRAM, and/or TCAM tables.

At 15 million packets per second, the IXP2800 provides for 64 read and 64 write SRAM references per packet. At 28 million packets per second, the IXP2800 provides for 32 read and 32 write SRAM references per packet. The references may be used for network address lookup, multi-tuple classification, policing, packet or buffer descriptors, queuing, statistics, and scheduling. The IXP2800 supports an aggregate rate in excess of 60 million queue operations per second on one or multiple queues. The number of queues that an IXP2800 supports is limited only by the available SRAM capacity, not by any on-chip resource limit. Tables 1 and 2 depict a possible allocation of SRAM bandwidth.

Table 1: Ingress possible allocation of SRAM references

| Function	QDR Reads	QDR Writes
Destination address TRIE route lookup	7	
7-tuple TCAM rule lookup	1	5
Buffer descriptor	2	2
Queue and freelist linked-list operations	6	6
Metering	3	3
Congestion avoidance (WRED)	5	4
Per-rule statistics	2	2
Per (min-size) packet totals	26	22

Table 2: Egress possible allocation of SRAM references

Function	QDR Reads	QDR Writes
Reassembly context	2	2
7-tupple TCAM rule lookup	1	5
Buffer descriptor	2	2
Queue and freelist linked-list operations	6	6
Congestion avoidance (WRED)	5	4
Per-rule statistics	2	2
Per (min-size) packet totals	18	21

Varying product requirements will increase or decrease the allocation of SRAM references per packet. For instance, incorporating ATM segmentation and reassembly into the processing flow will add a couple of read and write references on ingress and egress. A balanced system design will try to balance the consumption of resources across ingress and egress processors.

This configuration represents a cost-effective and extremely flexible approach to basic packet processing at 10Gb/s rates.

10GB/S MULTI-SERVICE SWITCH BLADE

In this system architecture, three IXP2800 network processors are used (Figure 2). The ingress processing is distributed across two IXP2800 network processors.
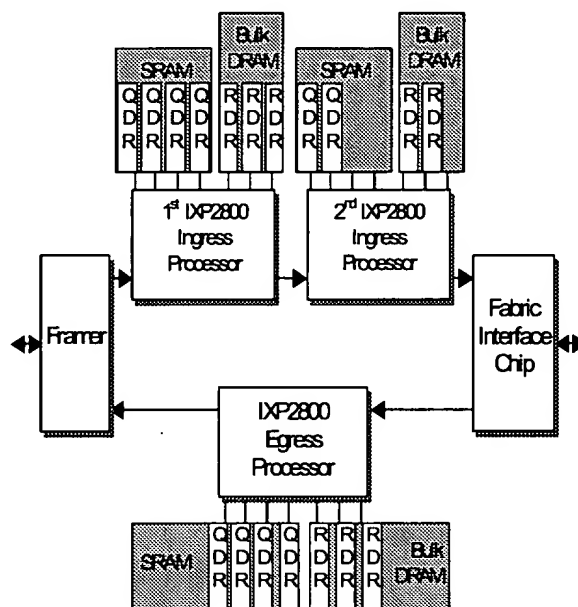
Figure 2: 10Gb/s multi-service configuration

The egress processing is accomplished with a single IXP2800, as in the prior configuration.

The distinguishing characteristic about this configuration is the division of labor between the 1st and 2nd ingress IXP2800 processors. The configuration is intended to support broadly varying rates of packet processing while maintaining expected aggregate throughput rates.

The 1st ingress IXP2800 is responsible for transferring received packet pieces into contiguous ring buffers in DRAM, as they are received, with minimal processing. Multiple rings may be supported, with the destination ring identified by minimal packet classification. These rings provide for an elasticity buffer to allow for varying rates of packet processing performed subsequent to the storage of the packets in DRAM. The size of the rings may vary, based upon the expected arrival rate of the packets and the elasticity requirements. Maintenance of the rings requires minimal SRAM accesses but does require static allocation of memory per ring. The smallest configuration of DRAM that supports the maximum bandwidth (3 DRAM components) supports 96 mega-bytes of storage.

Smaller rings in SRAM shadow the rings in DRAM. The entries in the SRAM rings provide status regarding the processing of the packet and information to pass on to the 2nd ingress IXP2800 for final processing. Upon completion of the processing of a packet in the 1st ingress IXP2800, the status is updated. Earlier packets may complete processing subsequently, but the ring insures in-order forwarding to the 2nd ingress IXP2800, as the packets are fetched from the ring in-order and only after they complete processing.

As packet-processing threads become available, a scheduling decision is made in software regarding which rings should be serviced. Threads read sufficient parts of packets in DRAM to process the packets. Threads may take an arbitrary time to complete processing of the packet, subject to the elasticity provided by the DRAM buffering and the average packet arrival rate. Separately, as Transmit Buffer (TBUF) elements become available, the status of the rings is polled to forward packets to the 2nd ingress IXP2800.

The 1st ingress IXP2800 is best suited to performing multi-level, multi-protocol packet classification and editing. By design, most of the SRAM bandwidth is available for classification. Update of flow-specific or queue-specific state, including statistics, is deferred until the 2nd ingress IXP2800. A digest is forwarded with the packet that describes such state as needs updating. (The bandwidth available through the SPI-4.2 physical interface approaches 20Gb/s, accommodating the increased payload per packet.)

The 2nd ingress IXP2800 receives packets with no packet interleaving or limited packet interleaving, reducing the accesses to SRAM to reassemble the packets. All classification processing has been completed. The 2nd ingress IXP2800 is responsible for any remaining metering and policing, statistics, queuing and buffering, congestion avoidance, and transmit scheduling into the fabric.

By design, the division of labor between the 1st and 2nd ingress processor distributes the use of SRAM bandwidth across the two processors. The 1st ingress IXP2800 supports nearly arbitrary processing times, while maintaining the order of packets within categories (rings). The 2nd ingress IXP2800 updates shared state in-order. The egress IXP2800 operates exactly as described in the prior configuration that also uses a single egress IXP2800.

OC-48 (4 X OC-12 OR 16 X OC-3) SWITCHING BLADE

In this system architecture, a single IXP2800 network processor is used for both ingress and egress processing (Figure 3). External silicon components multiplex the

data from the framer and fabric into the SPI-4.2/CSIX receiver and distribute the transmit data from the SPI-4.2/CSIX transmitter to the framer and fabric.

The IXP2800 supports the capability to simultaneously multiplex the SPI-4.2 and the CSIX-L1 protocols on the same interface. The switch chip allows interfacing to both an OC-48 framer (probably using SPI-3 or UTOPIA Level 3) and a fabric supporting a CSIX-L1 interface.
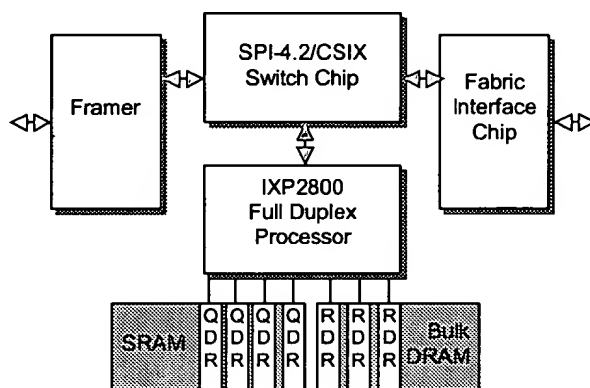

Figure 3: OC-48 (4 x OC-12 or 16 x OC-3) configuration

In this configuration, the IXP2800 is offered half of the aggregate load supported by the prior configurations. There is sufficient DRAM bandwidth to write packets to DRAM on receptions and read them back for processing as in the prior multi-service switching configuration, although the packets are stored using the linked-list organization of buffers. Rings of buffer descriptors are used to enforce in-order enqueuing of the packets to linked-list queues, just as in the prior configuration. The different code paths for ingress and egress processing may be handled on the same microengines or distributed across different microengines in order to optimize the utilization of the microcode stores. Finally, different microengines are allocated to updating shared state in-order and coherently.

THE IXP2800 MICROARCHITECTURE

The IXP2800 has 10 major internal units (Figure 4). The IXP2400 also has 10 major units; however, a few of the units have variations. The IXP2800 units and the variations for the IXP2400 are described below.

The Media-Switch-Fabric Interface

The Media and Switch Fabric (MSF) Interface is used to connect an IXP to a physical layer device (PHY) and/or a

---

Figure 4: IXP2800 block diagram
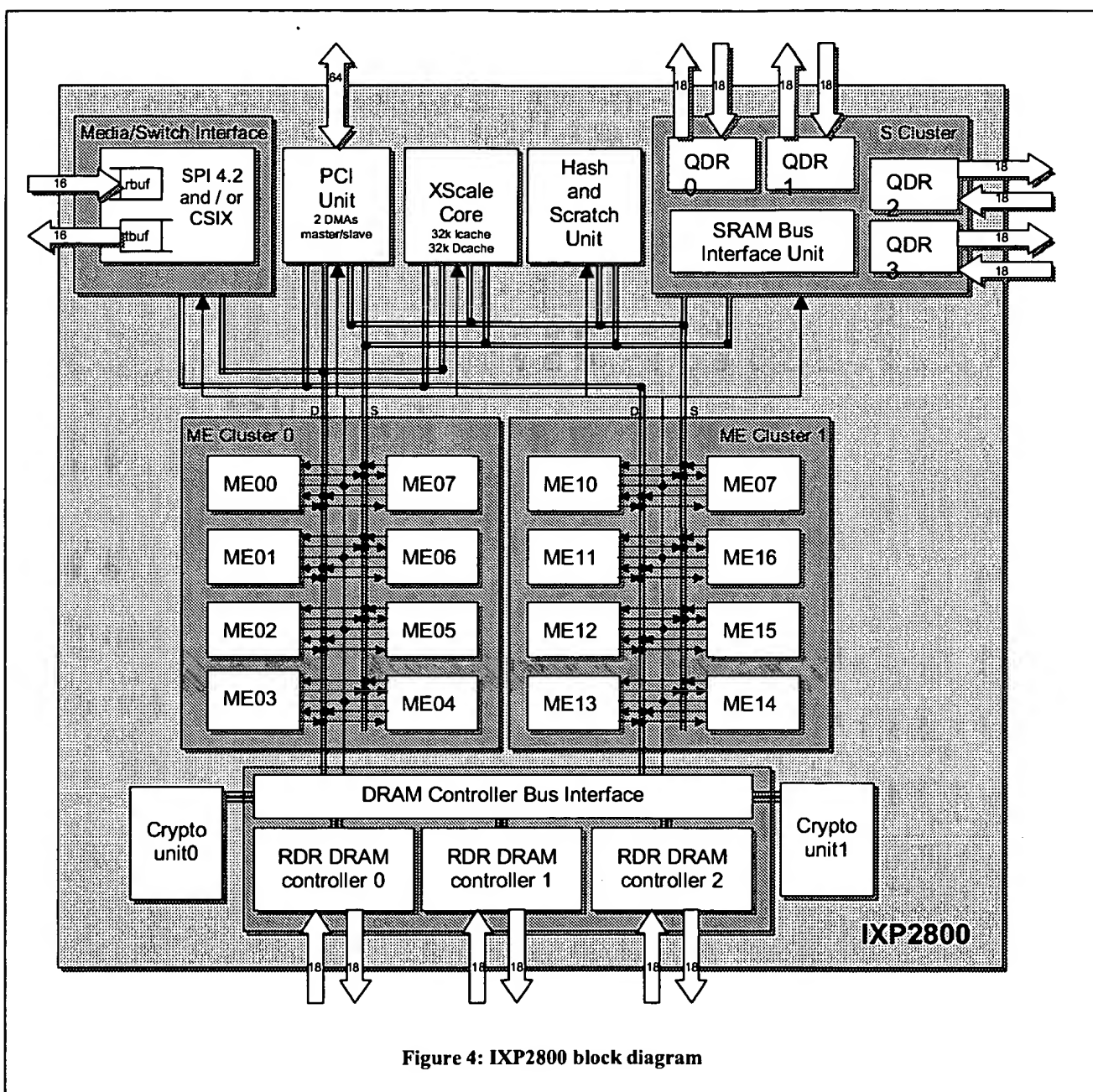
switch fabric. The MSF consists of separate receive and transmit interfaces. Each of the receive and transmit interfaces can be separately configured on the IXP2800 for either SPI-4 Phase 2 (System Packet Interface) for PHY devices or CSIX-L1 (Common Switch Interface Specification, Layer 1) protocol for switch fabric interfaces. Additionally, configuration provides for multiplexing both protocols over the interface simultaneously. The IXP2400 is similar; however, instead of SPI-4 phase 2 signaling and protocol, the IXP2400

supports POS PHY Level 3 (dual 32-bit uni-directional 125MHz bus) and CSIX-L1 protocol.

The receive and transmit ports are unidirectional and independent of each other. Each IXP2800 port has 16 data signals, a clock, a control signal, and a parity signal, all of which use Low Voltage Differential Signaling (LVDS) and are sampled on both edges of clock. There is also a flow control port consisting of a clock, data, parity, and ready status bits, and it is used to communicate

between two IXP2800 chips, or an IXP2800 and a switch fabric interface. All the high-speed LVDS interfaces support dynamic deskew training. The IXP2800 supports 10Gb/s inbound traffic and 15Gb/s outbound or 15Gb/s inbound and 10Gb/s outbound. The overspeed (15 vs. 10Gb/s) is required by fabrics, which have inherent inefficiencies. The average bandwidth required by a fabric may be 10Gb/s; however, for extended moments they may burst 15Gb/s. The IXP can source or sink these extended burst rates.

Incoming packets are received into the Receive Buffer (RBUF). Outgoing packets are held in the Transmit Buffer (TBUF). The RBUF and TBUF are both RAMs and store data in sub-blocks (referred to as *elements*), and are accessed by either the microengines or XScale™.

The RBUF and TBUF each contain 8KB of data. The element size is programmable as either 64 bytes, 128 bytes, or 256 bytes per element. In addition, either buffer can be programmed to be split into one, two, or three partitions, depending on application. For SPI-4, one partition is used. For CSIX, two partitions are used (control and data c-frames). For both SPI-4 and CSIX, three partitions are used.

The microengine can read data from the RBUF to the microengine in_bound registers using the MSF[read] instruction. The microengine can promote data from RBUF to DRAM directly using the DRAM[rbuf_rd] instruction.

The microengine can promote data into the TBUF along with status via writes from the outbound_transfer registers using the MSF[write] instruction. The microengine can control movement of data from DRAM directly to the TBUF using the DRAM[tbuf_wr] instruction.

The IXP Chassis

The chassis is the bus system, which interconnects all the units within the IXP. The chassis employs uni-directional buses to implement a microengine-based distributed memory storage mechanism. The microengine has inbound and outbound transfer registers. The chassis is used to retrieve data from the outbound transfer registers and deliver data to the inbound registers. The chassis consists of data busses, which connect the microengine transfer registers to the various shared resources (i.e., SRAM, DRAM, hash, cryptography units). Additionally, the chassis has multiple instantiations of a command bus. This command bus runs ahead of the data buses. It notifies the shared resources that a microengine is requiring service and indicates the source and destination addresses, the function to be performed, and any other information required to complete the requested task.

Additionally, the command bus has a field indicating the data length of the requested transfer.

The chassis operates at half the frequency of the microengine. This is up to 700MHz for the IXP2800 and up to 300MHz for the IXP2400.

THE MICROENGINE CLUSTERS

The IXP2800 has 16 microengines, configured as two clusters of eight identical microengines. The reason for this partitioning is to provide more communication capability between the microengine and the rest of the chip resources. Each cluster has its own copy of command and data busses. Thus each microengine shares the command bus with seven other microengines, rather than with 15 other microengines, as would be the case without the two-cluster configuration. More details about the capabilities and internal configuration of the microengine are presented later in this paper.

The SRAM cluster

The SRAM cluster consists of four independent SRAM controllers, each of which controls external Quad-Data-Rate (QDR) SRAMs. The reason for four channels is to provide sufficient control information bandwidth for 10Gb network applications. SRAMs are a good choice for control information, which tends to have many small data structures such as queue descriptors and linked lists. SRAMs, unlike DRAMs, allow for small access size and additionally allow access to any address sequence with no restrictions. Each SRAM controller, running at 200MHz, provides 800MB/s of read bandwidth and 800MB/s of write bandwidth.

In addition to the normal read and write access, the IXP2800 SRAM controllers provide three additional hardware functions.

1. *Atomic read-modify-write operations: increment, decrement, add, subtract, bit-set, bit-clear, and swap.* The atomic operations are useful for implementing software semaphores. They can also be used for multiple processes that modify a shared variable without using conventional mutex to obtain ownership, for example, update a network statistic via an atomic add operation. This is more efficient, since it eliminates the mutex operation altogether in this case.

2. *Linked-list queue operations.* This hardware accelerates enqueue and dequeue to linked-list operations by eliminating the read-to-write or read-to-read latency. For example, to do an enqueue, software must read the current list tail and then use it as an address to write the new link to memory. The SRAM controller keeps the tail address in on-chip registers and does the enqueue write locally; this saves the time that would have been spent by

the microengine to get the tail value and then simply use it as the address for the write.

3. *Ring operations.* A ring is also sometimes called a *circular buffer*. It consists of a block of SRAM addresses, which are referenced through a head and tail pointer. Data is inserted at the tail of the ring (using the content of the tail pointer as the address) and removed from the head (using the content of the head pointer as the address). The SRAM controller keeps the head and tail pointers in on-chip registers and increments them as they are used. The advantage is that multiple processors can add data to and remove data from the rings without having to use a mutex to obtain ownership.

It is also possible to attach an external coprocessor, such as Ternary Content Addressable Memory (TCAM), or classification processors to the SRAM interface. The interface conforms to the Network Processor Forum's LA-1 (Look-Aside) interface specification.

The DRAM Cluster

The DRAM cluster provides three independent DRAM controllers, each of which controls external Rambus DRAMs (RDRAMs). The reason for three channels is to provide sufficient data buffering bandwidth for 10Gb network applications. DRAMs are a good choice for a data buffer because they offer excellent burst bandwidth and are much denser and cheaper per bit relative to SRAM. Each DRAM controller, running at 133MHz (note that this equates to 533MHz DDR, which is 1066 M transfers/sec on the data pins), provides 17Gb/s of bandwidth, shared between reads and writes.

The three DRAM controllers provide hardware interleaving of the DRAM address space (often referred to as *striping*). This is done to spread accesses evenly to prevent "hot spots" in the memory. If all accesses for a period of time were to address only one of the controllers, then only one-third of the bandwidth would be available. The way the interleaving works is that each controller simultaneously receives all access requests and compares the address to the range of addresses that fall within its range. It then claims either all, part, or none of the access request according to the result of the address compare. The entire process is done in hardware, completely transparent to the software.

The Cryptography Unit

The cryptography unit performs authentication and bulk encryption. It is believed that these two datapath tasks are critical strategic functions for the network processor. The crypto engines are innovative designs that have a very small footprint, yet the two engines provide 10Gb/s throughput performance. This unit is covered in detail in a subsequent article in this journal.

The Hash Unit

The hash unit can perform either 48-bit, 64-bit, or 128-bit polynomial division. The hash function implemented is an irreducible polynomial, which has the characteristic of a one-to-one mapping. This means that if there is a collision, checking the unused bits of the remainder against that entry's saved and unused remainder bits confirms or denies the collision. The multiplier to the hash function is programmable so that if a default multiplier is not performing efficiently, a new one may be calculated.

The motivation for the hash unit hardware is that performing a high-quality hash in software is cycle consuming. Layer 2 lookups for Ethernet employ a hash on the 48-bit source and destination addresses for bridging. The hash hardware acceleration is excellent for this lookup. Ipv6 employs 128-bit source and destination addresses, and the hash unit may be used for data reduction.

The basic idea behind the hash unit is to take correlated data and uniformly distribute it across a small set space. For example, the hash unit may be used to take the 48-bit Ethernet destination address and map it into a much smaller 16-bit addressed destination table. A good hash function will uniformly distribute entries in the smaller table to reduce the probability of a collision.

The Scratch Unit

The scratch unit contains an on-chip 16KB scratchpad memory, running at 700MHz. To a programmer, the scratchpad memory provides very similar capability to the SRAM described earlier. The main difference is that the capacity of the scratchpad is much smaller than the external SRAMs. However, the scratchpad has lower latency (running at 700MHz instead of 200MHz as the external SRAMs). The scratchpad provides the atomic read-modify-write and ring operations as described in the SRAM section.

The XScale™ Processor

The XScale processor is compliant with the ARM Version 5TE (Advanced Risc Machines), and runs at 700MHz. Normally, it is used as a system control plane processor, handling exception packets and doing management tasks. It contains independent 32KB instruction and data caches, and a full capability memory management unit. The XScale has uniform access to all system resources, so it can efficiently communicate with the microengine though data structures in shared memory.

The PCI Unit

The PCI Unit provides an interface to industry standard 64-bit 66MHz PCI Rev 2.2. It is typically used as a control plane interface, either to an external microprocessor, for example, a Pentium®, or as an external device interface, such as a public key accelerator. The PCI unit can act as a PCI bus master, allowing XScale or microengine access to external PCI targets, or as a PCI bus target, allowing external devices to transfer data to and from the IXP2800 external SRAM and DRAM memory spaces. The PCI Unit also contains DMA channels that can be programmed to do bulk data transfers between DRAM and external PCI targets.

Pentium® is a registered trademark of Intel Corporation or its

subsidiaries in the United States and other countries.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

THE IXP2XXX MICROENGINE

Several goals guided the specification of the ME:

- Efficient silicon implementation. The need for lots of compute capability in the network processor dictated the need for a large number of MEs.

- High frequency to allow for sufficient instructions per packet. The ME has a six-stage pipeline and runs at 1.4GHz in P861 (.13
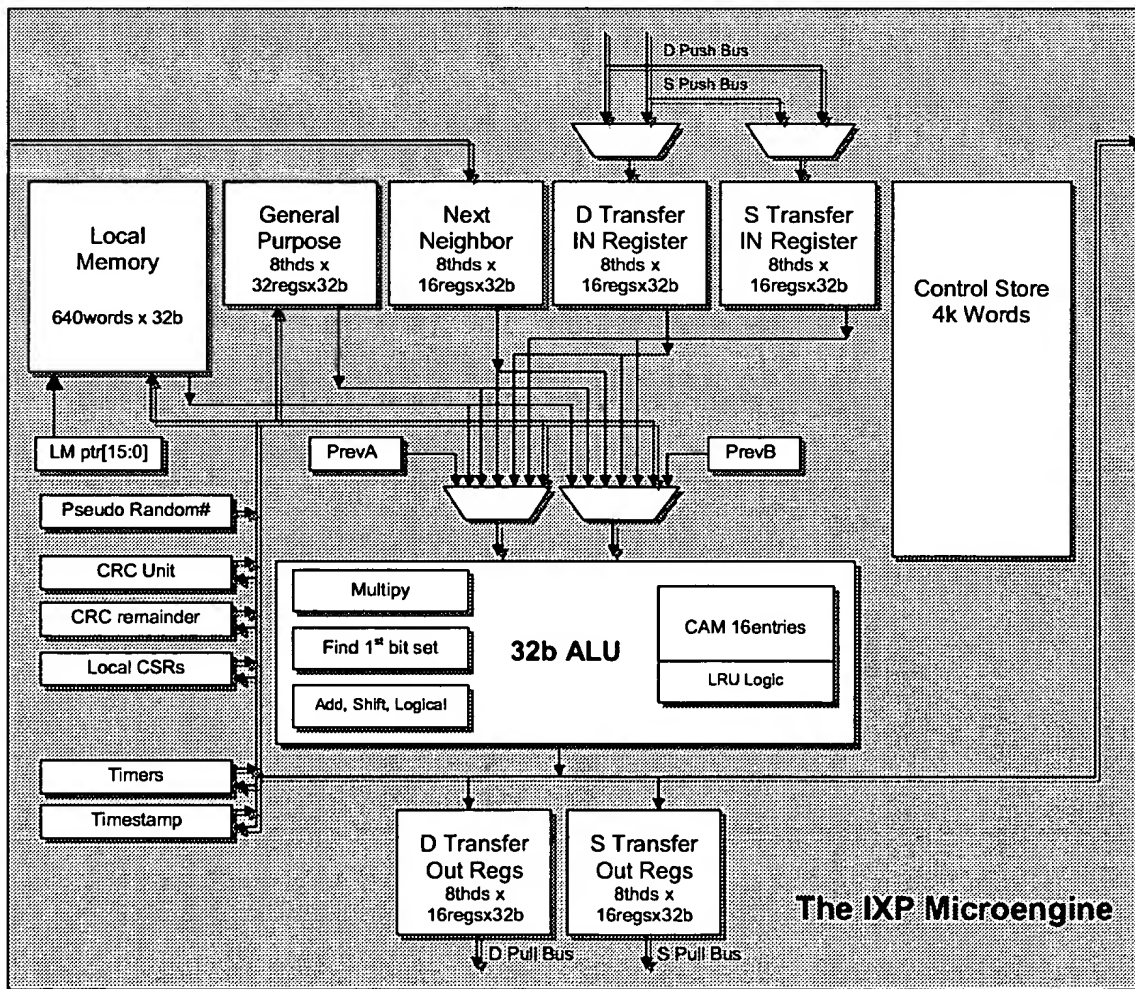

Figure 5: IXP microengine block diagram

micron).

- Large register set. Having many registers minimizes the need to shuffle program variables back and forth between registers and memory. Having to shuffle uses valuable cycles without accomplishing useful work.

- Low-latency local memory in the ME. This is addressable memory, in addition to the registers. It can be used in any way the application chooses, for example, to hold packet data or state related to ports, etc.

- Efficient intra-ME communication capability. This is useful in the applications described earlier in this article.

- Multiple threads. Given the disparity in processor cycle times vs. external memory times, a single thread of execution often blocks waiting for external memory operations to complete. Having multiple threads available allows for threads to interleave operation—there is often at least one thread ready to run while others are blocked. This makes more productive use of the other ME resources, which would otherwise be idle.

There are eight hardware threads available in the ME. To allow for efficient thread swapping, each thread has its own register set, program counter, and thread-specific local registers. Having a copy per thread eliminates the need to move thread-specific information to/from shared memory and ME registers for each swap. Fast thread swapping allows a thread to do computation while other threads wait for IO (typically, external memory accesses) to complete, or for a signal from another thread or hardware unit. (Note that a swap is similar to a taken branch in timing.)

Each of the eight threads will always be in one of four states.

- Inactive—Some applications may not require all eight threads. Unused threads can be kept in an inactive state by setting the appropriate value in a configuration register.

- Executing—The executing thread is the one in control of the ME. Its PC is used to fetch the instructions that are executed. A thread will stay in this state until it executes an instruction that causes it to go to sleep state (there is no hardware interrupt or pre-emption; thread swapping is completely under software control). At most, one thread can be in executing state at any time.

- Ready—In this state, a thread is ready to execute but is not because a different thread is executing. When the executing thread goes to sleep state, the MEs thread arbiter selects the next thread to go to the executing state from among all the threads in the ready state. The arbitration is round robin.

- Sleep—In this state, the thread is waiting for some external event(s) to occur (typically, but not limited to, an IO access). In this state the thread does not arbitrate to enter the executing state.

At most, one thread can be in executing state at a time; any number of threads can be in any of the other states.

Registers
As shown in the block diagram in Figure 5, each ME contains four types of 32-bit datapath registers:

1. 256 general-purpose registers

2. 512 transfer registers

3. 128 next neighbor registers

4. 640 32-bit words of local memory

Each of the first three types is partitioned per thread. The local memory is shared among all threads.

GPRs are used for general programming purposes. They are read and written exclusively under program control. GPRs, when used as a source in an instruction, supply operands to the execution datapath. When used as a destination in an instruction, they are written with the result of the execution datapath.

Transfer registers are used for transferring data to and from the ME and locations external to the ME (for example, DRAMs, SRAMs, etc).

Next Neighbor (NN) registers are used as an efficient method to pass data from one ME to the next, for example, when implementing a data-processing pipeline. The NN registers can supply instruction source operands; when NN register is the destination of an instruction, that value is written in the next ME.

The NN registers can also be configured to act as a circular ring instead of addressable registers. In this mode the source operands are "popped" from the head of the ring, and destination results are "pushed" to the tail of the ring. The head and tail pointers are maintained in hardware in the ME.

For applications that don't need to use the NN registers for intra-ME communications, the ME can be put into a mode where an instruction with NN as destination will write the NN register in the same ME. This increases the

number of registers available to an application. The choice of this mode is independent of the use of ring mode; all combinations are supported.

Local Memory (LM) is addressable storage located in the ME. LM is read and written exclusively under program control (i.e., it is private to the ME). The distinction between LM and the registers described above is that the LM address is computed by the program at run-time, whereas the register addresses are determined at compile time and bound in the instruction. Each thread has two LM address registers, which are written by special instructions. The specific LM location selected is based on the value in one of the LM address registers, which is specified in the instruction.

All of the registers described above, including LM, are built using two-ported register files: one read port and one write port. The area efficiency of two-ported registers relative to multiport registers is important in allowing the large number of registers to fit in the allocated silicon area. Of course, the use of two-port registers places some restrictions on which combinations of registers can source operands for each instruction. The restrictions are managed by the register allocator in the compiler and assembler, and in practice there are no limitations found in normal programs.

Instructions

The instruction set of the ME is similar to that of many RISC microprocessors, with some additional features tailored to the network processor task.

- Computation instructions can take one or two operands, perform an operation, and optionally write back a result. The sources and destinations can be GPRs, transfer registers, next neighbor registers, and local memory. The operations are shifts, add/subtract, logical, multiply, byte align, and find first one bit. There is also a Content-Addressable-Memory (CAM), described below.

- Logical operations can be performed along with shifting one of the operands in a single instruction. This can often be used to collapse two operations into one, for example, in masking fields of a header.

- IO instructions are used to read and write various memory units in the NPU, such as receive buffer, transmit buffer, DRAM, and SRAM. There are also a number of higher-level operations available in the IO units, such as ring operations, atomic read-modify-write, and linked-list queue operations.

- Special instructions are provided for inserting bytes into registers. These are useful for packet header modification.

- Branches can be done, based on comparing a byte within a register to a literal value. This can be used to efficiently test for values in a header. Branches can also be done on individual bits set or clear within a register. This is useful for efficiently testing status flags. The above are in addition to the normal suite of branches on numerical results, such as greater than, less than, etc.

- Instructions can be placed into branch defer slots to minimize the number of cycles lost due to taken branches redirecting the ME pipeline. The compiler is able to move instructions that are executed, regardless of branch outcome into those slots.

- Hardware support is provided for integer multiply. Each instruction cycle can retire 8 bits of operand. Taking this approach vs. providing a full, autonomous multiply was a trade off of performance vs. silicon area. One advantage of this approach is that for small numbers, for example, 8 bits or 16 bits, the compiler can insert just enough cycles to complete the multiply.

- Hardware support is also provided for CRC operations for several industry standard polynomial values. The hardware can do a CRC over 32 bits every other cycle. This is equivalent to 22.4Gb/s at a ME frequency of 1.4GHz.

CAM

The CAM is a unique function that has a number of uses. The CAM has 16 entries; and each entry stores a 32-bit value. This allows a source operand to be compared against 16 values in a single instruction. All entries are compared in parallel, and the result of the lookup is written into the destination register. There are two outcomes (the lookup result is indicated by the value in a destination register bit, which a branch instruction can test in one cycle):

- A *miss* indicates that the lookup value was not found in the CAM. The result also contains the entry number of the least recently used entry (which can be used as a suggested entry to replace).

- A *hit* indicates that the lookup value was found in the CAM. The result also contains the entry number that holds the lookup value. In addition,

the result holds an additional 4 bits of state that the program can define and use.

The CAM can be used to accelerate multi-way compares. It can also be used to act as the tag store of a cache; in this case, the entry number of a matching value can be used as an index to data associated with the value (and stored, for example, in SRAM or LM). Because the CAM does not store any of the associated data, the hardware places no limitation on the amount of data stored for each cached entry. It could be as little as a few bits or as much as needed, limited only by SRAM memory capacity. The state bits can be used to store additional information about a cache entry, for example, if it has been modified or how many threads are making use of it.

Event Signals

The ME supports the concept of event signals. These are signals that a thread can use to indicate the occurrence of some event external to the ME; the thread can block (go to sleep state) waiting on the event. Typical use of events includes completion of IO and signals from other threads, for example, to indicate that some data has arrived and is ready for processing. Each thread has 15 event signals. These can be allocated and scheduled by the compiler in much the same way as registers are allocated. They allow for a large number of outstanding events and, therefore, concurrent processing of non-dependent tasks. For example, the thread could start an IO to read packet data from the receive buffer, start another IO to allocate a buffer from a freelist, and start a third IO to read the next task from a work list (on a ring). All of the IOs execute in parallel. Many microprocessors can also schedule multiple outstanding IOs; normally, that is handled in a hardware-based scoreboard. By using event signals, the ME places much of the burden on the compiler, which simplifies the hardware.

Other microengine features useful to the network processor task are the following:

- o *Timestamp*–a 64-bit timestamp register that can be used for real-time tasks. The timestamp is guaranteed to be monotonically increasing for the lifetime of an application; it will not wrap around.

- o *Pseudo-random number*–used for some algorithms that need random numbers. Note that this is pseudo-random and not suitable for security applications.

CHALLENGES AT 10GB/S

For high-speed networking systems an extremely efficient means for handling successive enqueue and dequeue requests to the same linked list queue structure is required

to support a large number of queues (linked lists for memory efficiency) at line rate (packet/cell arrivals at ~40ns). Consecutive enqueue operations to the same linked list queue are latency constrained since the first enqueue must create the link to a list tail pointer before a subsequent entry can be linked on to that new tail. Likewise, for consecutive dequeue operations, the head pointer of the queue must be read to determine the new head pointer for the list before a subsequent dequeue operation is done. A control structure that can manage requests to a large number of queues as well as successive requests to only a few queues or to a single queue, plus a memory controller data path capable of back-to-back enqueue or dequeue to the same queue at the packet or cell arrival rate are required.

A single microengine designated the queue manager receives enqueue requests from the set of microengines that are programmed to perform receive processing and classification. The enqueue request specifies to which output queue an arriving packet or cell should be added. A microengine that functions as the transmit scheduler sends dequeue requests to the queue manager microengine that specifies the output queue from which a packet or cell is to be taken and then transmitted to an output interface (see Figure 6).

Each microengine contains a 16-entry Content Addressable Memory (CAM) that tracks which entry is the Least Recently Used (LRU). The queue manager microengine uses the CAM to implement a software-controlled cache containing the last 16 queue descriptors used to enqueue and/or dequeue packets or cells. While the CAM serves as the "tag store" holding the addresses of the queue descriptors that are being cached, the "data store" associated with each CAM entry is implemented in the SRAM controller logic. The data store for each queue descriptor contains the head pointer (address of the first entry of a queue), the tail pointer (address of the last entry of a queue), and a count entry (present "length" of the queue). Locating the data store for the cache of queue descriptors at the memory controller allows for low-latency access to and from the queue descriptor data cache and memory.

The queue manager microengine issues commands to return queue descriptors to memory and fetch new queue descriptors from memory such that the queue descriptor data store located at the memory controller remains coherent with the CAM tag store of queue descriptor addresses. The queue manager issues enqueue and dequeue commands indicating which of the 16 queue descriptor data store locations to use for the memory controller to perform the command.

All enqueue and dequeue commands are initiated in the order in which they arrived at the memory controller, and

these reference 1 of 16 data store tail or head pointers. An enqueue writes the address of the pointer to be added to the queue to the address of the cached tail pointer, and then updates the cached tail pointer to the address just added. Since enqueue requires only a write, the data store is updated in 2 cycles, and a subsequent enqueue even to the same queue can then be initiated. For dequeue, the address of the head pointer in the data store is returned to the queue manager microengine (this is the address locator for the buffer or cell to be transmitted), and a read of the contents of the head pointer is initiated. When the read data returns, it is loaded into the head pointer for specified data store entry. A subsequent dequeue request to a different queue can be initiated on the next cycle. However, a dequeue request to a queue where a read of the head pointer location is in progress must be held up until the data store location for that entry's head pointer is updated. An enqueue to a queue with a dequeue in progress can proceed since the tail pointer is not affected by the dequeue.

Having the control structure for queueing in a microengine allows for flexible high performance while using the existing hardware of the microengine. Distributing the data store part of the cache of queue descriptors allows for the low-latency memory operations required for successive enqueue and dequeue operations at high line rates.
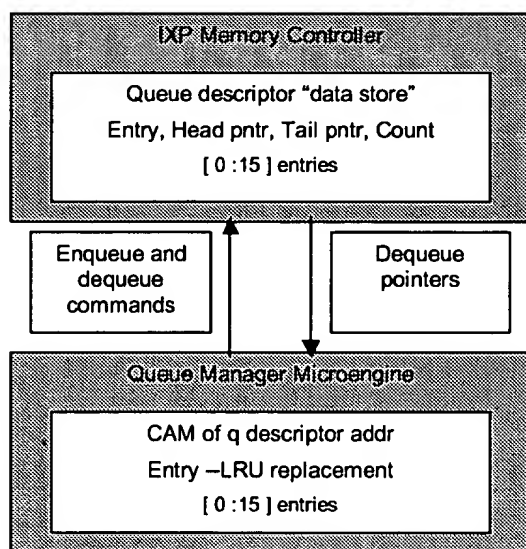

Figure 6: Enqueue dequeue memory controller

DISCUSSION

There is a trade off between programmable flexibility and software complexity. Flexibility provides great product

advantages for feature enhancements and future upgrade capabilities. However, it also makes evaluation for performance against customer requirements and subsequent customer product development more challenging. The IXP family is addressing these challenges with tools and leadership silicon performance.

The IXP workbench is a state-of-the-art integrated development environment. Users write their code, compile (C language) or assemble (IXP macro language) their code with advanced error reporting, then debug the code on a very high-performance-cycle accurate simulator (>500 cycles per second simulation performance). This simulation environment provides advanced visualization tools and debugging facilities for rapid code maturation. The workbench environment can then be used to exercise the IXP silicon with the developed code. Advances to the workbench include rapid prototyping and static performance evaluation, given simple user heuristics.

Providing leadership silicon performance requires less software tuning to achieve given product goals. The IXP2800 with 16 parallel processors at 1.4GHz delivers on the promise of network processors. This promise includes providing a multi-application hardware-based platform for communication companies to leverage across multiple market segments. Additionally, it promises network processor customers differentiation by software. Within a given company, the promise of common software routines or functions to be leveraged by different product groups is also now possible for IXP customers.

CONCLUSIONS

The IXP family provides a very powerful, flexible hardware platform for a wide range of software-based network processing applications. The range of applications is widening and is identifying the opportunity for certain IXP variations tuned to specific applications. Recognizing this possibility, the design and implementation methods for the IXP family have been optimized for rapid future variations.

This is enabling a roadmap vision that is two-pronged. One prong is providing greater performance through the use of additional hardware multi-threading and additional microengines, while also including new strategic hardware acceleration engines such as the IXP2800 did with advanced dataplane cryptography acceleration.

The second prong is leveraging the Intel Communication Group's silicon portfolio for greater system integration. Integration is important when it can reduce system power, cost, and board area. This prong can provide current IXP customers with a product cost-reduction path.

Both of these prongs will leverage the silicon capabilities afforded by 90nm and, subsequently, 65nm high-performance CMOS.

Performance, integration, advanced tools, rapid software prototyping, advanced strategic hardware acceleration, extreme customer support: this is the roadmap vision for the IXP family.

ACKNOWLEDGMENTS

The authors acknowledge the contributions of Sanjeev Jain, David Romano, John Cyr, Jim Guilford, Bob Kushlis, Jose Niell, Milo Sprague, Kin-Yip Liu, Yim Pun, John Wishneusky, Donald Hooper, Bill Wheeler and the VMOD development team, John Sweeney, the IXP verification teams, and the IXP implementation teams led by John Beck (IXP2800) and Ahmad Zaidi (IXP2400).

REFERENCES
[1] Matthew Adiletta, et. al, "Packet over SONET: An Overview of the Packet Processing Flow of a 10 Gigabit/sec Datastream Mapped to an IXP2800," *Intel Technology Journal*, Vol. 6 Issue 3, August 2002.

[2] Internet Network Working Group, RFC 2697, September 1999.

[3] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM* Transactions on Networking, V.1 N.4, August 1993, pp. 397-413.

[4] Internet Network Working Group, RFC 2863, April 1998.

[5] E. Johnson and A. Kunze, *IXP1200 Programming*, Intel Press, ISBN 0-9702846-7-5, 2002.

AUTHORS' BIOGRAPHIES
Matthew Adiletta is an Intel Fellow and Director of Communication Processor Architecture. He led the architectural development and implementation of the IXP2800 and is driving the IXP roadmap. He is interested in processor architecture and advanced implementation techniques for rapid silicon development. He is also intrigued with network security and classification. Adiletta has been responsible for 12 previous silicon chips, including silicon for VAXes, alphas, video, graphics, and communication. The IX2800 is the lucky 13th. Adiletta received his B.S. degree in electrical engineering, with Honors, at the University of Connecticut. He resides in Bolton, Massachusetts. His e-mail address is matthew.Adiletta@intel.com.

Debra Bernstein is an architect for Intel's Network Processor Division. She worked on the architecture of the IXP2000 series and the IXP1200. For the 2000 series, Deb has been particularly focused on the queuing problem. Previously, she worked on microprocessors in the VAX and Alpha family at Digital Equipment Corporation. She is a 1982 graduate from the University of Massachusetts at Amherst. Her e-mail address is debra.bernstein@intel.com.

Mark Rosenbluth is an architect in the Network Processor Division. He has been at Intel for four years and prior to that worked at Digital Equipment Corporation, where he was architect for PCI Bridges and also worked on VAX and Alpha microprocessors. He received a B.S.E.E. degree from Rutgers University. He resides in Uxbridge, Massachusetts, and can be reached via e-mail at mark.rosenbluth@intel.com.

Hugh Wilkinson is a systems architect in the Network Processor Division at Intel. Hugh's technical interests include switching fabrics, protocol design, software decomposition, and high-speed signaling. He received his B.S. degree in Computer Science from Boston University. He works in Hudson, Massachusetts, and can be reached at Hugh.Wilkinson@intel.com.

Gilbert Wolrich is a senior architect in the Network Processor Group in Hudson. He has contributed to the definition of both the IXP1200 and IXP2000 solutions. Gil has worked on high-performance network and general-purpose processors, numerous floating point units, and is interested in network security. Gil received a B.S. degree from R.P.I. and an M.S. degree from Northeastern University in Electrical Engineering. He resides in Framingham, Massachusetts, and can be reached via e-mail at gilbert.wolrich@intel.com.

Network Processor Performance Analysis Methodology

Sridhar Lakshmanamurthy, Kin-Yip Liu, Yim Pun,
Larry Huston, Uday Naik
Intel Communications Group, Intel Corporation

Index words: network processors, IXA, OC-48, MEv2, line-rate performance, IP DiffServ, benchmark, IXP2400

ABSTRACT

This paper describes the performance analysis methodology developed to analyze the performance of various networking applications that are targeted for running on the IXP2400 network processor, the second-generation IXA network processor.

Traditionally, CPU benchmarks and system-level benchmarks have been used to understand the performance of general-purpose computer systems. However, such standards are still evolving in the field of network processors. Furthermore, network processors are targeted to diverse applications, and their performance is intricately tied to both the hardware features implemented on-chip and the software running on them, posing significant challenges in developing and using standard benchmarks.

The methodology described in this paper addresses the challenges in analyzing the performance of various networking applications running on the IXP2400 network processor. This methodology involves dividing the application into pipeline blocks, estimating the compute and IO requirements for each block, estimating the available processing and latency budget for each pipeline element, and mapping the application blocks to the software paradigms and the hardware resources. This methodology is validated by writing and tuning the microcode blocks for the application.

This paper also describes a case study using the IPv4 forwarding + DiffServ application running on the IXP2400 to analyze and demonstrate OC-48 line-rate performance for a 46B minimum-sized POS packet.

INTRODUCTION

Network processors are an emerging class of chips that are highly programmable and optimized for processing packets at wire speed. Specifically, these processors are designed to handle deep packet inspection that spans layers 3 through 7 of the Open Systems Interconnection (OSI) network model and are targeted for applications in the OC1 (55Mb/s) to OC192 (10Gb/s) data rates. Flexibility and programmability make the network processor a good candidate to replace expensive and inflexible ASIC chips for all the fast-path (data plane) processing in network equipment, as these network processors (NPU) can cover a wider range of applications. This provides faster time-to-market, increased flexibility, and lower costs to network equipment Original Equipment Manufacturers (OEMs).

A key challenge in making NPUs successful is establishing their performance capabilities. Traditional CPU benchmarks such as the SPEC CPU2000 [1] suite (comprised of the CINT2000 for integer benchmarks and CFP2000 for floating point benchmarks) have been used extensively to understand the performance of general-purpose CPUs. For benchmarking computer systems, the Transaction Processing Performance Council [2] has developed system-level benchmarks such as TPC-C (On-Line Transaction Processing benchmark), TPC-H (ad-hoc decision support benchmark), and TPC-W (transactional web e-commence benchmark). However, such standards are still evolving in the field of network processors. Furthermore, network processors are targeted to diverse applications, and their performance is intricately tied to both the hardware features and the software running on them, posing significant challenges in developing and using standard benchmarks. This paper describes a methodology that addresses the challenges involved in analyzing the performance of networking applications running on the IXP2400 network processor and presents a case study using the IPv4 forwarding + DiffServ application.

A key ingredient of the performance analysis methodology is a detailed data movement model of the

target application. This model describes the various operations performed by the network processor on every received packet. Depending on the application, these operations could include protocol header error checks, payload error checks, route lookup, flow identification, complex rule-based filtering, header/payload compression/encryption, policing, congestion management, queuing, scheduling, rate shaping, rate limiting, and transmitting.

The next step of the methodology uses the data movement model to estimate the number of compute cycles and total I/O references required for these operations on a per-packet basis. The total compute cycles are determined by the number of instructions that need to be executed for completing the necessary tasks. The I/O references include all operations to external memories to read and write information required during the processing stage. High-level pseudo-code is developed to arrive at these estimates.

Another key ingredient of the methodology is the estimation of the total available budget for packet processing. This budget is determined based on the packet inter-arrival time, and depends on the network processor frequency, the data rate at which packets are received by the network processor, and the smallest packet size that could be received by the processor.

The results of the above estimation efforts, namely,

1. available budget per-stage based on the processor frequency, data rate and the minimum packet size and

2. total compute and I/O cycles required by the application

determine how the functional blocks are mapped onto the available h/w resources (microengine contexts, on-chip scratch memory, external DRAM and SRAM) and how the software concepts (hyper-task chaining, pool of threads, functional pipeline, context pipeline) are used to meet the performance goals.

The methodology is validated by implementing microcode and tuning the code on the simulator and the hardware to demonstrate line-rate performance.

This paper uses the above outlined methodology to demonstrate OC-48 line-rate performance for 46B POS minimum-sized packets for the IPv4 forwarding + DiffServ application running on IXP2400. The following sections provide a brief overview of the internal and external architecture of IXP2400, describe the data movement model for the IPv4 forwarding + DiffServ application, and discuss the performance analysis and tuning of this application.

IXP2400 NPU OVERVIEW

Figure 1 shows the external interfaces of the IXP2400 NPU. The IXP2400 has two 32-bit interfaces to move network data in and out of the chip. The RX and TX interfaces support industry standard protocols for data movement such as SPI3, POS-PHY-L2 for packet-based interface, Utopia 1,2,3 for cell-based interface, and CSIX protocol for the switch fabric interface [3,4,5].
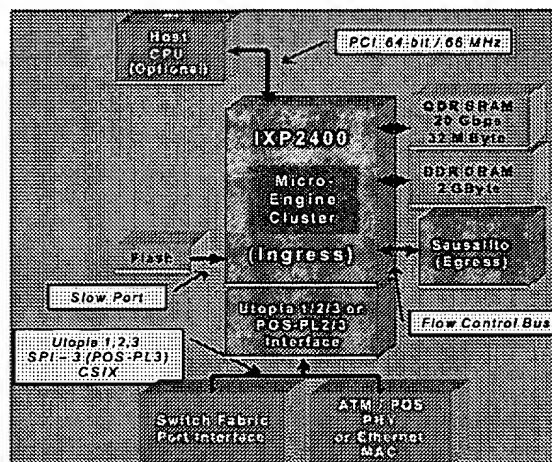

Figure 1: IXP2400 external interfaces

This interface can be independently configured to be 1x32, 2x16, 4x8, or 2x8+1x16 and can be clocked at 25MHz-125MHz, providing full flexibility to use the processor in any application ranging from OC-3 to OC-48 data rates. At 125MHz, the interface provides a peak bandwidth of 4Gb/s in and out of the chip to support the overhead of switch fabric encapsulation.

Since each IXP2400 provides only half-duplex OC-48 connectivity, two such chips are necessary for a full-duplex line card. To support this configuration, the IXP2400 also has a 4b/8b CSIX flow control bus that is used to communicate fabric flow control information between the two processors. At 125MHz, this interface provides up to 1Gb/s of peak bandwidth for flow control messages.

The IXP2400 has one channel of industry standard DDR DRAM running at 150/300MHz, providing 19.2Gb/s of peak DRAM bandwidth. The channel can support up to 2GB of DRAM. The DRAM is primarily used to buffer packets.

In addition to the DRAM, the IXP2400 also provides two channels of industry standard QDR SRAM running at

200/400MHz, providing 12.8Gb/s of read bandwidth and 12.8Gb/s of write bandwidth. Up to 32MB of SRAM can be populated on the two channels. The SRAM is primarily used for packet descriptors, queue descriptors, counters, and other data structures.

The NPU can communicate with the host processor over the 64b, 66MHz PCI. The slow port interface is used to connect to the Flash memory and also provides the general-purpose IO interface.
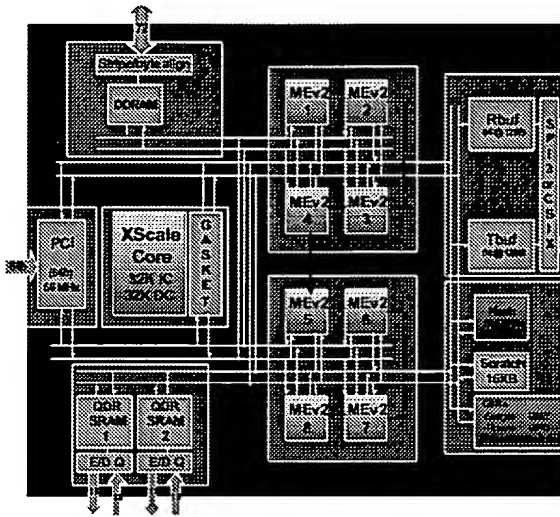

Figure 2: IXP2400 internal architecture

Figure 2 shows the internal architecture of the IXP2400. IXP2400 contains eight multi-threaded, packet-processing microengines. These eight microengines are highly programmable packet processors and support multi-threading of up to eight threads each. Each microengine provides a variety of network processing functions in hardware and provides the ability to process data at OC-48 wire-speed. IXP2400 also offers extensive communication mechanisms between all on-chip processing units and enables the microengines to readily form different topologies of software pipelines that can be customized for various target applications and network traffic patterns. The memory controllers facilitate efficient accesses to the off-chip SRAM and DRAM.

The IXP2400 microengine design includes additional features to increase performance and simplify development. These new features include the following:

- A multiplier for quality of service (QoS) algorithms such as metering and traffic shaping.

- A pseudo-random number generator to accelerate congestion avoidance algorithms like Weighted Random Early Discard (WRED).

- Cyclic Redundancy Check (CRC) hardware that verifies and generates Cyclic Redundancy Code (CRC) for Asynchronous Transfer Mode (ATM), ATM Adaptation Layer 5 (AAL5), Ethernet, Frame Relay, and High-Level Data Link Control (HDLC) protocols.

- 16-entry Content Addressable Memory (CAM) used to implement a data cache in local memory. The CAM facilitates efficient data sharing among microengine threads, resulting in greater performance, as well as reduced consumption of precious memory bandwidth.

- A 64-bit local timer with programmable time-out signaling to enhance traffic scheduling and shaping.

- 640 words (4B) of local memory that is shared by all the threads.

The IXP2400 also has an integrated low-power general-purpose Intel® XScale™ microarchitecture core. The integrated XScale processor offers ample processing power for running control plane software.

IXP2400 also offers a variety of low-latency communication mechanisms among the microengines and the integrated XScale processor. These communication mechanisms consist of dedicated high-speed data-paths between neighboring microengines, data-paths between all microengines, shared on-chip scratchpad memory, and shared First-In-First-Out (FIFO) ring buffers in scratchpad memory and SRAM. These innovations enable the microengines to form various topologies of software pipelines flexibly and efficiently, allowing processing to be tuned to specific applications and traffic patterns. This combination of programming flexibility and efficient inter-process communication ensures performance headroom while minimizing processing latency.

Network processing applications typically need to perform extensive queue management. Depending on the applications and algorithms used, the network processor may manage thousands of packet queues. The network processor must execute the desired scheduling algorithm and select the appropriate packets out of these queues for transmission at wire speed. As a result, effective queue management is key to high-performance network processing and to reducing development complexity. The IXP2400 provides high-performance queue management hardware that automates adding data to and removing data from queues. Multiple threads can access queues simultaneously. The size of each queue and the number of queues is limited only by the amount of memory available.

Pentium® is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

IXP2400-BASED LINE CARD CONFIGURATION

Figure 3 shows a full-duplex OC-48 line card configuration using the IXP2400. The two IXP2400 processors labeled ingress and egress processors execute the IPv4 forwarding + DiffServ application described in the next section.
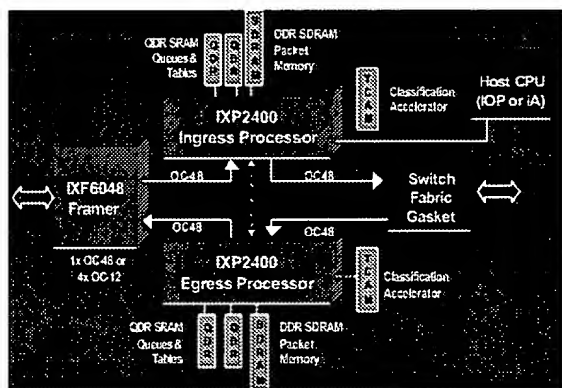

Figure 3: IXP2400-based OC-48 line card configuration

PERFORMANCE BUDGET

A key metric required for the performance analysis is to estimate the available budget. The available compute budget is calculated on a per microengine basis and determines how much processing the network processor can perform on each packet. The available IO latency budget is determined by the number of threads used in a processing block.

The compute budget for a given application is determined by the size of the smallest packet that needs to be processed and the targeted data rate. In other words, the packet inter-arrival time determines how many cycles are available for processing each packet. In order to keep up with the arrival rate, no processing stage in the pipeline can exceed this inter-arrival time.

On the POS interface, the smallest IP packet that can be transferred is 40B (20B of IP header and 20B of TCP header). Assuming a PPP protocol overhead of 6B, the total size of the minimum POS packet becomes 46B. At an

OC-48 data rate of 2.5Gbps, the inter-arrival time between two back-to-back minimum-sized POS packets is 147ns.

The following equation shows the relationship between the various parameters in estimating this packet inter-arrival time:

$$PacketInterArrivalTimeIn(ns) = (PacketSizeInBytes*8)/(DataRateInGbps)$$

The packet inter-arrival time measured in nano-seconds (ns) is obtained by dividing the packet size specified in bits with the desired data rate where the units for the data rate are specified as gigabits per second.

The time in nano-seconds can be converted into processor clocks using the following formula:

$$PacketArrivalTimeInProcessorClocks = PacketInterArrivalTimeIn(ns)/ProcessorClockTickIn(ns)$$

The processor clock tick is the reciprocal of the processor frequency. For example, if a processor is running at 100MHz, the processor clock tick equals 1/100MHz or 10ns. Similarly, 600MHz processor frequency translates to a processor clock tick of 1/600MHz or 1.67ns.

Assuming a 600MHz clock on the MicroEngines (MEs) in the IXP2400, the minimum packet inter-arrival time of 147ns translates to 88 microengine cycles. In order to keep up with the arrival rate, any processing stage of the pipeline must complete all the required processing for a given packet within this budget and should be able to process a new packet every 88 cycles.

The compute cycle requirements for other data rates or minimum-sized packets can be derived in a similar fashion. For example, ATM cells have a fixed size of 53B. At the OC-48 data rate, back-to-back ATM cells arrive every 170ns or 102 microengine cycles. This is the available budget per microengine for processing ATM cells.

Reducing the data rate requirement or increasing the packet size increases the available compute cycles per ME. For example, the total available budget for handling POS minimum packets at the OC-12 data rate (622Mbps) is 355 cycles per ME, four times the available budget compared to OC-48 since the data rate of OC-12 is lower than the OC-48 data rate by a factor of four. Similarly, the above equations can be used to calculate the available budget for handling 100B POS packets at OC-48 data rate. This equals 192 cycles per ME.

Figure 4 shows the relationship between the data rate and the packet size on the available budget per ME in terms of available ME cycles. As indicated above, the available budget increases as the packet size increases or the data rate decreases.
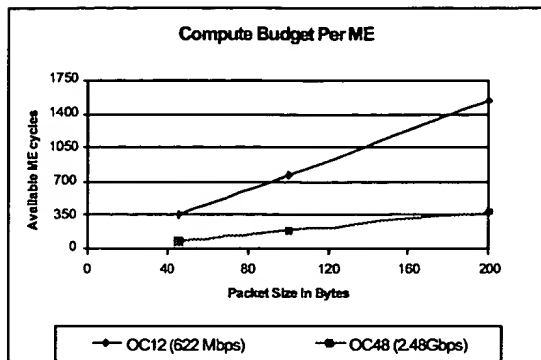
Figure 4: ME compute budget estimate

As the IXP2400 has 8 microengines, the total available compute cycles for the entire application is 8 times the total available budget per ME. In other words, the total available compute cycles on the ingress and egress IXP2400 for the IP DiffServ application at OC-48 data rates for handling minimum POS packets is 8*88 or 704 cycles.

Packet processing involves accesses to internal and external memories such as scratch memory, SRAM, DRAM, etc. The number of memory accesses per stage depends on the data movement model for that stage. Typically, the latency to access memory is several minimum packet arrival times, i.e., SRAM access latency would be 150 cycles (almost twice the POS minimum packet arrival time), while DRAM access latency would be 250-300 cycles (>3 times the POS minimum packet arrival time). The software-controlled multi-threaded features on the ME provide the mechanism to hide these large latencies. Using 8 threads on an ME provides an IO latency budget of 8 times the packet arrival rate. In the above situation of 150-cycle SRAM latency, a total budget of 8*88 or 704 cycles (8 times the POS minimum packet arrival time) allows the stage to support four dependent SRAM operations. Assuming 250-cycle DRAM latency, each stage can support two dependent DRAM operations and still maintain line rate.

In the multi-threaded implementation, each thread within the ME is assigned a new packet that arrives into the system. Assuming M MEs and N threads per ME, a total of $M*N$ packets can be handled in parallel by these multi-threaded engines before the first thread of the first ME needs to be ready to process the $M*N$ +1th packet. This provides a mechanism to increase the total available compute and IO budget.

Typically if M is 1, the software pipeline implementation is referred to as a context pipeline. $M > 1$ implies that several MEs are grouped together to handle a given processing block. Such a pipeline is referred to as a functional pipeline. The following paragraph describes these software pipeline models supported by the IXP2400.

The IXP2000 programming model provides two types of software pipelining models:

- A *context pipeline*, in which different pipeline stages are mapped to different MEs. Each ME constitutes a context pipe-stage. A packet context is passed from one pipe-stage to the next using the various inter-ME communication mechanisms. The available compute budget per context pipeline stage is the same as the budget available per ME.

- A *functional pipeline*, in which a packet context remains within an ME while different functions are performed on the packet as the time progresses. The ME execution time is divided into n pipe-stages and each pipe-stage performs a different function. Multiple MEs are assigned to the functional pipeline to increase the available compute and IO times. For example, if all 8 threads of 4 MEs are used in a functional pipeline, the total compute budget for the minimum POS packet will be 4*88 = 352 cycles, and the total IO latency budget will be 4*88*8 = 2816.

The total compute and total IO operations required for a given block determine the pipeline that would be used for that stage.

IPV4 FORWARDING + DIFFSERV APPLICATION

The line card configuration uses the SPI3 [3] interface to receive Internet Protocol (IP) packets from the SONET framer. This mode is also known as the Packet over SONET mode (POS). The fabric interface for the line card uses Common Switch Interface (CSIX) protocol [5], standardized by the Network Processor Forum.

The data movement model definition for this application involves identifying all the processing blocks that are executed for each packet.

The ingress IXP2400 processor receives POS frames that carry IP payload. Since the IXP2400 supports up to 16 logical ports on the framer, the IP packet segments can arrive interleaved. The first pipeline stage reassembles these segments into complete IP packets and stores the

packets into DRAM. In the subsequent pipe-stages, the ingress processor performs the following tasks:

- Route lookup to determine the next hop forwarding information by executing a Longest Prefix Match (LPM) algorithm on the destination IP address.

- RFC 1812 [6] compliant IP header checks to validate the IP header.

IP packet classification into flows and queues using either a 5-tuple or a 7-tuple lookup, i.e., classification based on IP source and destination address, Transmission Control Protocol (TCP) source and destination ports, protocol field, L2 port, etc.

- Execution of a Single-Rate Three-Color Marker (SrTCM) [7] meter pipeline stage to meter the traffic on a per-flow basis and mark the packet as green, yellow, or red, based on the flow parameters and arrival rate.

- Execution of a congestion avoidance algorithm such as Weighted Random Early Discard (WRED) that will randomly drop packets when the queue lengths exceed certain thresholds with the goal of minimizing congestion in the fabric.

Another challenge in obtaining OC-48 performance is the ability to add and delete packets from the queue at twice the packet arrival rate and still support a large number of queues. This is achieved on the IXP2400 by using the on-chip high-performance queue management hardware.

The transmit pipeline includes fully programmable schedulers such as Weighted Round Robin to schedule traffic into the fabric, and a transmit engine that adds fabric encapsulation to the IP frame, segments the IP frame into CSIX c-frames, moves c-frame data from memory and

to the transmit buffers, and enables data transmission on the CSIX interface.

The data movement model for the egress processor is constructed using a similar methodology. The first pipeline stage of the egress processor receives the CSIX c-frames and reassembles the original IP payload using the fabric encapsulation information. Subsequent packet processing stages of the egress processor perform further classification, if required, and execute metering and congestion avoidance algorithms. Sophisticated scheduling algorithms such as Deficit Round Robin are implemented on this processor to provide quality of service (QoS) for the network-bound traffic. The final transmit stage of the data movement model segments the IP frame into transmit chunks called mpackets and enables data transmission on the SPI3 interface.

Similar data movement models can be defined for analyzing the performance of other applications such as Asynchronous Transfer Mode (ATM), Segmentation and Reassembly (SAR), ATM traffic management, voice over ATM etc. The next section of this paper describes how this data movement model is used to estimate the cycle count and I/O requirements for each processing stage.

PERFORMANCE ANALYSIS METHODOLOGY

Figure 5 shows the ME allocation and the software pipeline chosen for each of the blocks in the ingress data flow for the IPv4 forwarding + DiffServ application. This partition is based on the cycle count estimates and IO requirements that are derived from the pseudo-code that corresponds to the data movement model described above. The subsequent sections provide the details on the pseudo-code-based analysis and the choice of the pipeline for each of these stages.
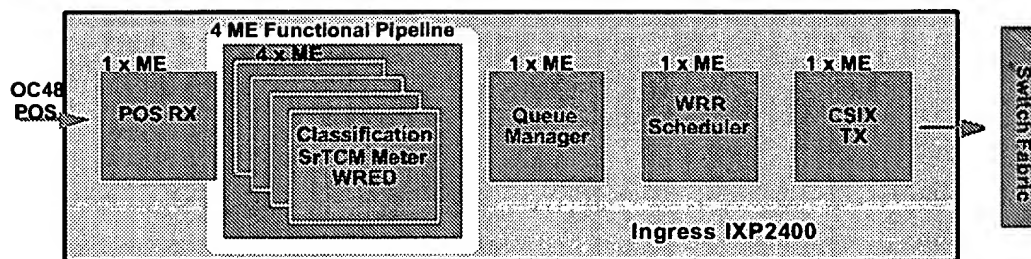

Figure 5: ME allocation for IP DiffServ application

Table 1: Pseudo-code for reassembly stage

Description	Cycle Count	Command	SRAM Rd	SRAM Wr	DRAM Rd	DRAM Wr
The receive thread wakes up, looks at the receive status word (RSW), and decides what to do with the rmpacket, based on SOP/EOP indicator within the RSW.	10					
For all conditions (SOP/EOP) move the RBUF data to DRAM. ME updates the reassembly context with the new DRAM location and offset.	30	1				64
For minimum-sized packets (and for EOP), update the buffer descriptor in SRAM	11	1		4		
For EOP, Write SOP ptr, EOP ptr, L2 port, Next IP hop, Class, Payload length to scratch ring for the next pipeline stage	11	1				
Prefetch new buffer descriptor to be used in the next phase	2	1	4			
Clean up, return the thread back to the thread freelist, free RBUF element	6	2				
Total for frame reassembly	70	6	4	4	0	64

POS Receive Block

The first block in the ingress flow is responsible for reassembly of the POS packets. On receiving a new packet, each thread in this block checks the Start of Packet (SOP) and End of Packet (EOP) bits of the packet, identifies the port of the packet, allocates a DRAM buffer for the packet on start of a new packet or when the previous buffer is full, updates the reassembly context with the current offset in the DRAM buffer, moves the data from the receive buffer to the DRAM buffer, signals the next stage of the pipeline on EOP, and cleans up the state for the next round.

This stage requires the following four IO operations: 1) SRAM read to allocate a new buffer; 2) DRAM write to move the packet data into the DRAM buffer; 3) SRAM write to update the packet descriptor information; and 4) a scratch ring write to signal the next pipeline stage when the entire packet has been reassembled with the packet data.

The pseudo-code for this pipeline stage is captured in an excel format and is shown in Table 1. Based on this pseudo-code, the reassembly block for POS minimum packets requires approximately 70 compute cycles and 4 IO operations. Since each ME has an 88-cycle compute budget for handling POS minimum packets, the reassembly function meets the cycle count budget for 1 ME. The next step in the analysis involves estimating the total IP latency for the reassembly block to determine conformance to the 1 ME budget. Assuming 8 threads of the ME are used, the total available IO budget is 704 cycles. Assuming the latency for the SRAM and scratch operations is 125 cycles each, and the latency for the DRAM operation is 250 cycles, the total latency for the 2 SRAM ops + 1 scratch operation + 1 DRAM operation is 2*125 + 1*125+ 1*250 is 625 cycles. This fits within the 704-cycle available budget.

Thus, based on the pseudo-code analysis and the latency estimate, the POS receive block is implemented as a context pipeline running on all 8 threads of a single ME.

Since the latency estimates used in this analysis are approximate, establishing the actual performance of this block requires successive refinement and tuning of the actual microcode running on the IXP2400-cycle-accurate simulator. This simulation environment provides the dynamic latency value for each of the IO operations based on the total load on the system. The code is tuned such that the dynamic latency budget still fits within the total available IO latency budget.

During the tuning effort, if the dynamic latencies exceed the available budget, other tricks could be used to fit within the budget. The available options include handling multiple packets per thread such that the total available IO budget is further increased. For example, if each thread handles two packets concurrently (with each packet in a different phase of execution), the total available IO budget increases to 2*704 cycles. However, the flip side of handling multiple packets per thread is the increase in the actual cycle count per thread. Thus, a balance is required to determine whether the compute budget or the IO budget becomes the bottleneck.

In cases where the compute budget exceeds the available cycles on the single ME, the context pipeline stage will be required to be converted into a functional pipeline stage, with additional MEs added to provide the necessary headroom.

Classification Pipeline

The classification stage reads the message from the previous reassembly stage and reads the IP/TCP header of the packet from DRAM. This stage performs routing and classification functions using the information in the header.

One example of packet classification used in the DiffServ application is a 5-tuple exact match lookup to identify the flow and queue ID of the packet. The fields used for the 5-tuple search are IP source and destination addresses, the TCP source and destination ports, and the protocol field. Since the 5-tuple lookup uses 104 bits, a hash scheme is used to efficiently store the necessary information in SRAM. The hash key is generated using the hash hardware on chip. The hash key is used to access SRAM. On completion of the read, the original key is compared to the retrieved key. When a hash collision occurs, multiple dependent SRAM read operations are required to get the flow and queue information.

Once the packet is classified, this pipeline stage also executes all the checks mandated by RFC1812 including decrementing the time-to-live field of the IP header and updating the IP header checksum.

The pseudo-code-based analysis for the classification stage shows that this stage requires 2 DRAM operations plus 9-10 SRAM/scratch operations. Assuming similar latency estimates that are used in the POS Rx analysis, this block requires a total IO latency budget of approximately 1750 cycles, more than the total available IO budget for 2 MEs. Initial pseudo-code-based cycle count estimates showed approximately 160 cycles for performing the 5-tuple exact match and for executing all the header checks mandated by RFC1812, within the budget for a 2 ME functional pipeline.

The SrTCM meter block requires approximately 80 compute cycles and needs only 2 SRAM operations, reads the meter parameters for the given flow, and updates the relevant field of the flow data structure based on the behavior of the current packet. Similarly, the WRED block requires approximately 80 compute cycles and needs only 2 SRAM operations.

Since the classifier block requires more IO budget than the available budget on 2 MEs, while the meter and WRED blocks have a lot of IO budget headroom, the entire pipeline is implemented as a 4 ME functional pipeline, whereby the classification stage can use up the IO budget allocated and unused by the other processing blocks.

Thus, the pseudo-code-based analysis allows the application blocks to be partitioned appropriately to maximize the use of all available resources.

Each thread on each ME handles one packet; thus a total of 32 packets remains active in this pipeline, providing a total latency budget of 4*8*88 = 2816 cycles to retire each packet. In other words, each thread receives a new minimum packet every 2816 cycles. In order to keep up with the OC-48 line rate, each thread in the classification stage must retire the previous packet within 2816 cycles.

Queue Manager

The Queue Manager (QM) is responsible for performing enqueue and dequeue operations on the transmit queues for all packets. The functionality of the QM is identical on both the ingress and egress processors: to process enqueue and dequeue requests from the other pipe-stages and perform the necessary operations on the queue array structures. The enqueue operation does not return any data. The dequeue operation returns a pointer to the buffer descriptor that was dequeued. This information is transferred to the ME in the transmit pipeline via a scratch ring. It is possible to request a dequeue from the QM either before a schedule decision is made or after a schedule decision is made. This is implementation specific and depends on the number of queues supported, scheduling algorithm used, etc. On the ingress processor, the dequeue request is issued after the schedule decision, while on the egress processor the dequeue request is issued before the schedule decision

The pseudo-code-based analysis for the QM shows that each enqueue and dequeue operation requires 4 SRAM/scratch operations each. This stage does not require any DRAM operations. Each thread handles 1 enqueue operation and 1 dequeue operation in parallel; thus the IO accesses required for the enqueue and dequeue operations can be issued concurrently. Thus, the total IO latency is determined by 4 dependent SRAM operations. Assuming 125-cycle latency, this total latency of 500 cycles fits within the latency budget for 1 ME. Thus the QM is implemented as a context pipe-stage running on 1 microengine.

CSIX Transmit Scheduler

The Common Switch Interface (CSIX) scheduler schedules packets to be transmitted to the CSIX fabric. The scheduling algorithm implemented is Round Robin among the ports on the fabric and optionally Weighted Round Robin among the queues on a port. The scheduling and transmit is done a c-frame at a time.

The CSIX scheduler handles

o Flow control messages from the fabric. These messages are sent by the fabric to the egress IXP2400, which sends them on the c-bus to the ingress IXP2400. If the fabric asserts Xoff on a particular Virtual Output Queue (VoQ), the scheduler stops scheduling for the queue.

o Queue transition messages from the queue manager. A queue is scheduled only if the queue has data.

o MSF Transmit State Machine. The scheduler monitors how many packet c-frames are in the pipeline, and if

the number of packets in the pipeline exceeds a certain threshold, the scheduler stops scheduling.

For both the VoQ status and the transmit queue status, the scheduler keeps hierarchical bit vectors and uses the MEv2 Find First Bit Set (FFS) instruction to scan them efficiently. During each loop, the scheduler

o Determines if the TX pipeline is within the threshold.

o Picks up from where it left off in the last iteration and finds the next bit set and determines which queue to schedule.

o Sends a dequeue message to the queue manager to dequeue the head of that queue. The queue manager dequeues a cell (c-frame) from the head of the queue and sends a transmit request on a scratch ring to the CSIX TX microblock.

Pseudo-code analysis of the scheduler block shows that this block is compute intensive. In the worst-case condition, the scheduler exceeds the 88-cycle budget if it has to process fabric flow control messages, process QM queue transition messages, check for MSF transmit count, and schedule a c-frame at each minimum packet arrival slot. However, the worst-case condition is not likely to occur often, and by slowing down the processing task, the scheduler alleviates some of the problems of the worst-case condition, i.e., fabric flow control messages would slow down if the scheduler slowed down the transmit requests to the fabric, and QM queues would build up, resulting in fewer queue transitions from 1->0 or 0->1 if the scheduler falls behind.

The scheduler block has negligible IO operations. Thus, the CSIX scheduler block is implemented as a context pipe-stage executing on a single ME using only 4 threads. One thread of this ME executes the actual scheduling algorithm. Three support threads handle the fabric flow control messages, the QM queue transition messages, and the MSF transmit counter.

CSIX Transmit

The CSIX transmit engine handles the data movement from DRAM to the transmit buffers. This block receives transmit request messages from the queue manager. For each transmit request, a c-frame is transferred into a Transmit Buffer (TBUF), which is then transmitted into the fabric by the MSF transmit state machine.

Every request has an associated packet, which is being segmented into c-frames. The associated segmentation state for the packet and the packet meta-data is cached in local memory and is looked up using the CAM. The TX microblock adds the CSIX header onto the c-frame along with the packet data. Along with the CSIX header, a

Traffic Manager (TM) header is also added per c-frame, carrying extra information (destination layer-2 port ID, input blade ID, sequence number, next-hop ID, etc.) about the packet to be passed to the Egress IXP2400. In addition, the flow ID, class ID, input port, and some other fields from the meta-data are passed along to the Egress IXP2400 using a per-packet header pre-pended to the start of the first c-frame of each packet.

Pseudo-code-based analysis of the CSIX transmit block shows that this stage requires 3-4 SRAM/scratch operations plus 1 DRAM operation for scheduling each c-frame. For minimum POS packets, the latency budget is tight. Tricks such as concurrently handling 2 c-frames per thread are used to increase the total IO latency budget such that this block fits as a context pipe-stage that runs on a single microengine.

CONCLUSION

Analyzing the performance of network processors poses major challenges since these chips are targeted to a wide range of applications. A consistent methodology is required to establish the performance capabilities of these processors. This paper described a methodology for analyzing the performance of diverse networking applications that are targeted for the IXP2400 network processor. This methodology involves estimation of the available processing and latency budget per packet, estimation of the total compute and IO operations required per packet for the various pipeline stages of the target application, and mapping the pipeline stages of the application to the available hardware resources on the IXP2400 and utilizing all the available software pipelining techniques to hit the desired performance. The paper also presented a case study using the IPv4 forwarding + DiffServ application to validate the above methodology. This performance analysis methodology can be easily extended to evaluate the performance of other applications running on the IXP2400 network processor or to evaluate the performance of other network processors. The results from this methodology helped establish the performance capabilities of the IXP2400 network processor for various edge and access router applications.

ACKNOWLEDGMENTS

REFERENCES

[1] http://www.spec.org/ Standard Performance Evaluation Corporation.

[2] http://www.tpc.org/ Transaction Processing Performance Council.

[3] http://www.oiforum.com/ Optical Internetworking Forum.

[4] http://www.atmforum.org/ ATM Forum.

[5] http://www.npforum.org/ Network Processor Forum.

[6] http://www.ietf.org/rfc/rfc1812.txt?number=1812 IETF RFC specifying requirements for IP version 4 routers.

[7] http://www.ietf.org/rfc/rfc2697.txt?number=2697 IETF RFC specifying single-rate three-color marker.

[8] http://developer.intel.com/design/network/products/npfamily

AUTHORS' BIOGRAPHIES

Sridhar Lakshmanamurthy is a senior staff architect at Intel's Network Processor Division, focusing on understanding edge/access networking applications, analyzing the performance of Intel's network processor solutions, and defining future enhancements to these solutions. Prior to joining the Network Processor Division, Sridhar focused on platform performance analysis for Intel server chipsets for Xeon & Itanium Product Family (IPF) processors, and system bus specification for the IPF processors. Sridhar joined Intel in 1993 after receiving an M.S. degree in Computer Engineering from Rice University in Houston, TX. He also holds a B.E. degree in Electronics and Communication Engineering from Osmania University, Hyderabad, India. He can be reached via e-mail at sridhar.lakshmanamurthy@intel.com

Kin-Yip Liu co-manages the NPD NPBU Architecture team at San Jose, focusing on network processors for the Access and Edge market segments. His prior assignments include engineering and management positions in the Itanium Product Family architecture and firmware teams and in the 386SL and 486SL microprocessor projects. Kin-Yip Liu joined Intel in 1990 after completing his Master of Engineering (EE), Bachelor of Science (EE), and Bachelor of Arts (Economics) degrees from Cornell University. Kin-Yip holds four US patents in microprocessor architecture. His technical interests include network processing, computer architecture, and simulator development. His e-mail address is kin-yip.liu@intel.com

Yim Pun is the chief architect for the IXP2400 network processor. His technical interests include network processor architecture, microprocessor architecture, and network-processor-based system solutions for the switching/routing/ATM/MPLS/IPsec/SSL/VoIP applications in the metro/access/edge markets. He received his Master of Engineering degree from Cornell University in 1987, and a Bachelor of Electrical Engineering-Computer Engineering degree from the University of Wisconsin-Madison in 1986. He can be reached via e-mail at yim.pun@intel.com

Larry Huston is a principal software architect at Intel's Network Processor Division. He is responsible for defining the software requirements for future network processors as well as designing the advanced programming framework. Prior to Intel, Larry was a software architect at NetBoost where he helped design their programming environment for accelerating network applications such as firewalls and intrusion detection. Prior to NetBoost, Larry was a member of the technical staff at Ipsilon Networks where he designed and implemented Ipsilon's protocols for distributed IP switching and forwarding. Larry received his Ph.D. degree in Computer Engineering from the University of Michigan in 1995. He also holds M.S.E. and B.S.E. degrees in Computer Engineering and Aerospace Engineering from the University of Michigan. Larry can be reached via e-mail at larry.huston@intel.com

Uday Naik is a senior staff software engineer at Intel's Network Processor Division. His professional interests include networking, embedded systems, and digital television. Uday received his Master's degree in Computer Science from the University of Indiana, Bloomington in 1992. He also holds a Bachelor's degree in Computer Science and Engineering from the Indian Institute of Technology (IIT), Bombay. Uday resides in San Jose, California, and can be reached via e-mail at uday.naik@intel.com

Packet over SONET: Achieving 10 Gigabit/sec Packet Processing with an IXP2800

Matthew Adiletta, Donald Hooper, Myles Wilde
Intel Communications Group, Intel Corporation

ABSTRACT

The IXP2800 is the high-end device of a family of network processors developed by Intel Corporation. It is designed for 10 Gigabit/sec data rates, with typical usage in packet forwarding systems. It can be configured with large amounts of dynamic and static storage for buffering hundreds of thousands of packets for up to a million Internet Transmission Control Protocol (TCP) connections. The programmability and parallel nature of this processor chip make it an ideal choice when high performance and ability to quickly adapt to new network standards are the requirements.

This paper describes the evolution of an OC-192 (10 Gigabit/sec) design for Packet over SONET (POS), using the IXP2800. This was a particularly difficult challenge due to the fact that the arrival rate of packets is one per 40nS (which equals 57 microengine processor cycles). At this rate, for each packet, buffers must be allocated, the packet must be received, reassembled, and stored in DRAM, header verified, multi-field and destination lookups performed, packet classified for destination, timestamp saved, packet tagged (metered) by priority, previous header stripped, IP header modified, evaluated as to whether it should be dropped, statistic counters updated, enqueued for transmit, new fabric header prepended, transmitted, and buffers freed. This paper chronicles the issues encountered and solutions devised to achieve high packet-forwarding rates. The resulting architectural concepts of context pipe stages, functional pipe stages, phase interleaving, critical sections, elasticity buffers, and pool of threads are defined. The techniques of next-neighbor message passing, scratch rings, signaling, CAM state caching, local memory link-lists, SRAM link-lists, and reflected mailbox messages are explored. The OC-192 POS pipe stages are described.

A performance summary is reported for a variety of packet streams.

INTRODUCTION

The IXP2800 started with a simple marketing requirement: Achieve a high-end device that supports 10 Gigabit/sec data forwarding rates and is scalable to much higher rates when configured with a switch fabric.

In parallel with developing the hardware architecture, the IXP2800 design group decided also to develop a proof-of-concept application to explore how programs could be developed on this chip. Packet over SONET was chosen, as this presented the most difficulty in achieving full wire rate. When used over SONET, the minimum IP packet size is 40 bytes. A PPP layer 2 encapsulation is used with this, which adds another 9 bytes (4-byte header, 5-byte trailer), for a total 49B minimum packet size. By comparison, the minimum packet size for Ethernet is 64 bytes with an interpacket gap of 20 byte times.

We assume the reader is familiar with the Intel Network Processor Family [1].

CHALLENGES AND SOLUTIONS

Several obstacles had to be overcome to achieve the required performance. The following sections describe each problem and a corresponding solution.

Context Pipe Stage

Some of the operations on packets are well defined, with minimal interface to other functions or strict order implementation. Examples include update-of-packet-state information, such as the current address of packet data in a DRAM buffer for sequential segments of a packet, updating linked list pointers while enqueuing/dequeuing

for transmit, and policing or marking packets of a connection flow. In these cases the operations can be performed within the 57-cycle stage budget. Further difficulty arises in keeping operations on successive packets in strict order and at the same time achieving cycle budget across many stages. A block of code performing this type of functionality is called a *context pipe stage*.

In a context pipeline, different functions are performed on different Microengines (MEs) as time progresses and the packet context is passed between the functions or MEs. Each ME constitutes a context pipe stage (Figure 1). Cascading two or more context pipe stages constitutes a context pipeline. The name *context pipeline* is derived from the observation that it is the context that moves through the pipeline.
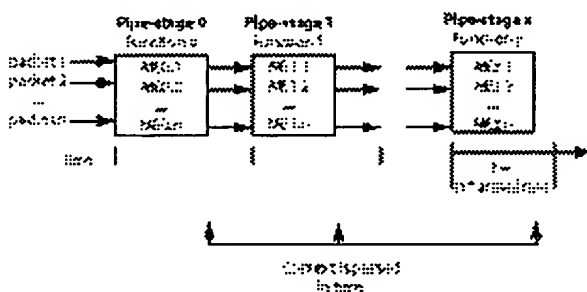

Figure 1: Context stages

A context pipeline is ideally suited when either the program store required to maintain this pipe-stage pipeline function is large or the data set associated with this function is large, or both. Additionally, it is appropriate when the data set associated with the function is greater than the data set associated with the context. An example is the data set for a transmit scheduler: keeping in the local memory up-to-date prioritized link-lists containing state information for active transmit queues. Such state information could include queue cell counts and flow control status.

Storing the information locally enables a function to efficiently maintain state, whereas the context for this pipe stage concerns which queue is being tasked with transmit. In this case it is obvious that the queue number (context) is much less than the data associated with the link lists (function data set).

Another advantage of the context pipeline is that the entire ME program memory space can be dedicated to a single function. This is important when a function supports many variations that result in a large program memory footprint. Cases in which the context pipeline is not desirable are ones in which the amount of context

passed to and from the pipe stage is so large that it affects system performance.

Each thread in an ME is assigned a packet, and each performs the same function but on different packets. As packets arrive, they are assigned to the ME threads in strict order. There are eight threads typically assigned in an IXP2800 ME context pipe stage. Each of the eight packets assigned to the eight threads must complete its first pipe stage within the arrival rate of all eight packets.
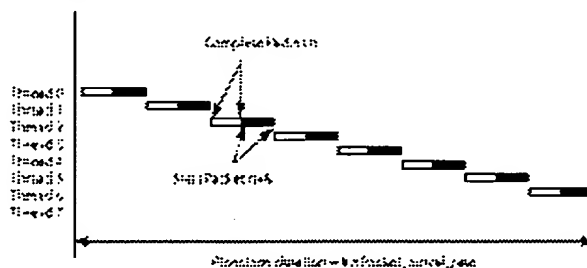

Figure 2: Interleaved phased piping

A more advanced context pipelining technique is shown in Figure 2. This technique interleaves multiple packets on the same thread, spaced eight packets apart. An example would be ME0.1 completing pipe-stage 0 work on packet 1, while starting pipe-stage 0 work on packet 9. Similarly, ME0.2 would be working on packet 2 and 10. In effect, 16 packets would be processed in a pipe stage at one time. Pipe-stage 0 must still advance every 8-packet arrival rates. The advantage of interleaving is that memory latency is covered by a complete 8 packet arrival rate.

Functional Pipe Stage

Another problem arises when the size of the packet state information passed between functions is so large that it is very costly (in cycles) to pass this information between MEs.
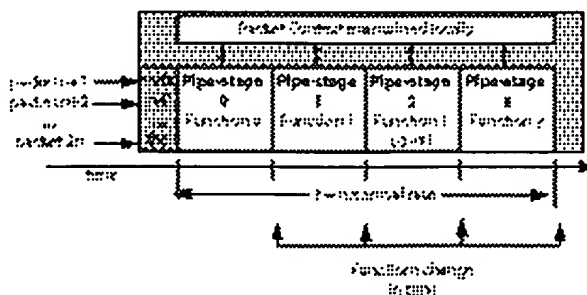

Figure 3: Functional stages

Context pipes can also be wasteful of instruction cycles if the functions being implemented are extremely simple.

In these cases, blocks of code can be arranged in what is known as a *functional pipeline*, a name derived from the observation that functions mo ve through the pipeline.

In a functional pipeline, the context remains with an ME while different functions are performed on the packet as time progresses. The ME execution time is divided into n pipe stages, and each pipe stage performs a different function. As with the context pipeline, packets are assigned to the ME threads in strict order.

There is little benefit to dividing a single ME execution time into functional pipe stages. The real benefit comes from having more than one ME execute the same functional pipeline in parallel. Figure 3 shows four functional pipe stages distributed across 4 MEs, each ME executing with eight threads.

A packet remains with a thread for a longer period of time as more MEs are added to the functional pipe stage. In this example, the packet remains with a thread-32 packet arrival time (8 threads x 4 MEs) because thread ME0.0 is not required to accept another packet until all the other threads get their packets.

The number of pipe stages is equal to the number of MEs in the pipeline. This ensures that a particular pipe stage executes only in one ME at any one time. This is required to provide a pipeline model that supports critical sections. A critical section is one in which an ME thread is provided exclusive access to a resource (such as CRC residue, reassembly context, or a statistic) in external memory. Critical sections are described in more detail later.

Functions can be distributed across one or more pipe stages; however, the exclusive access to resources as described above cannot be supported in these pipe stages.

The goal when designing a functional pipeline is to identify critical sections and place them into their own pipe stages. The non-critical sections of code then naturally fall into pipe stages that become interleaved with the critical sections. Non-critical code that takes longer than a pipe-stage time to execute must be allocated to more than one pipe stage.

The advantages of functional pipelines are these:

1. Unlike the context pipeline, there is no need to pass the context between each pipe stage, since it remains locally within the ME.

2. Functional pipelines support a longer execution period than context pipe stages.

The disadvantages of functional pipelines are these:

1. The entire ME program memory space must support multiple functions.

2. The latency of the functional pipeline is fixed at the worst-case path through any sequence of functions. If this time is exceeded, the packet must either be dropped or handed off to a slow path (e.g., to Xscale™) to complete packet processing.

Mixed Pipelines

Mixed pipelines employ elasticity buffers between pipelines. Figure 4 shows a context pipeline feeding a ring elasticity buffer that feeds a functional pipeline. The Functional pipeline subsequently feeds another ring elasticity buffer and finally a context pipeline.
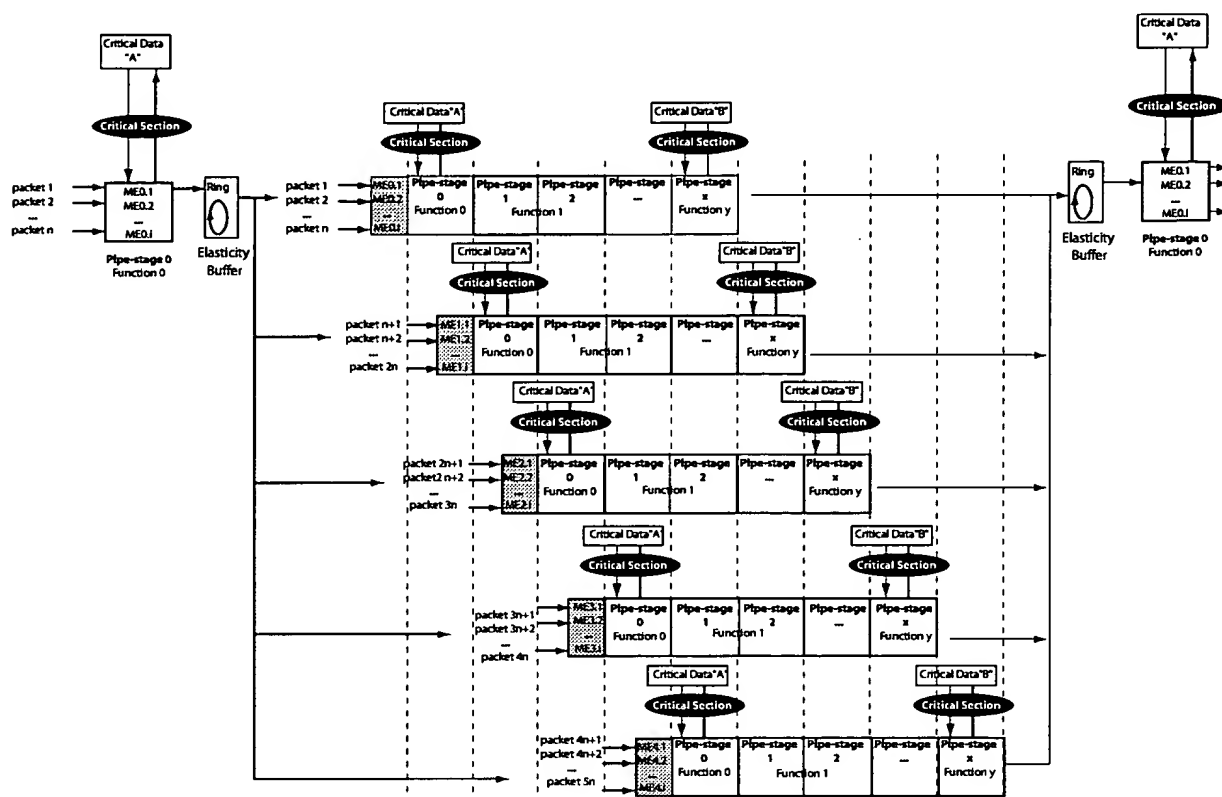
Figure 4: Ring elasticity buffers

Elasticity Buffers

When a pipeline transition occurs, pipeline lock-step execution is not required, and an elasticity buffer (implemented as a ring) can be used. Each pipeline must keep up with line rate. Each pipeline may have multiple pipe stages. The pipeline writing the ring is the producer. The pipeline reading the ring is the consumer. The IXP2800 has hardware assist for maintaining producer consumer rings. The problem to be solved is that there may be single or many producers, and single or many consumers and packet order for both producer and consumer must be maintained. The hardware assist includes hardware-maintained head and tail pointers for multiple rings in either scratch or SRAM memory. When a single producer communicates with a single consumer, Next Neighbor Register Rings can be employed. This is a very low latency private data path.

A pipeline that contains many producers (a multi-ME functional pipeline) can use software techniques in order to maintain order. MEv2 provides a Generalized Thread Signaling (GTS) mechanism for optimized order maintenance among the threads (defined below).

The advantage of using elasticity buffers is twofold. By decoupling producer/consumer heartbeats, independent

software development teams can develop different pipe stages independently and connect them using an API to the ring. Secondly, ring buffers allow short-term line rate processing anomalies/jitter to be hidden.

## Synch Section Signaling and Critical Signaling

In a functional pipeline, an ME should not transition into a critical section pipe-stage unless it can be assured that the previous ME has transitioned out of that critical section. In addition, the previous critical section must make sure its write data has progressed such that the next pipe stage reading the data gets the latest coherent data. This can be accomplished by placing a fence around the critical section using inter-thread signaling. There are four ways to signal another thread using inter-thread signaling, as listed in Table 1.

Table 1: Interthread signaling methods

| Thread-Signaling Method	Mechanism
Signal next thread in the same ME	Local csr write
Signal a specific thread in the same ME	Local csr write
Signal the thread in the next or previous ME	Local csr write
Signal any thread in an ME	CSR write to CSR Access Proxy Unit (CAP)

Synch Sections

A synch section is a write-dominated section of code in which multiple writers to the shared resource must be sequenced correctly. This is the feather-passing problem. In order to provide strict thread sequencing, a feather is passed from one thread to the next, and a thread can only begin execution when it owns the feather (Figure 5). The IXP2800 uses General Thread Signalling (GTS) to pass the feather. An example of a synch section is code that writes to a ring for enqueue requests while maintaining packet order. The next thread does not have a read conflict with the prior thread, but it must not write to the ring before the prior thread completes its write.
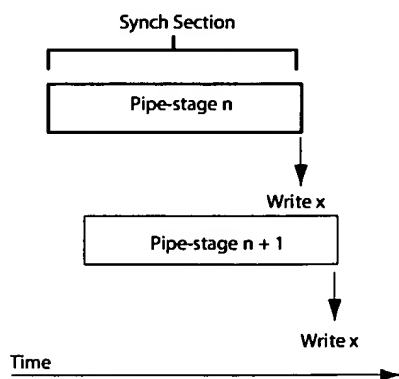
Figure 5: Synch section

Critical Sections

A critical section is a section of code that has only one ME thread with exclusive modification privileges for a global resource (such as a location in memory) at any one time (Figure 6). This is to protect coherency during read-modify-write operations. The following discussion focuses on providing exclusive modification privileges

between the MEs and providing exclusive access between the threads in an ME.
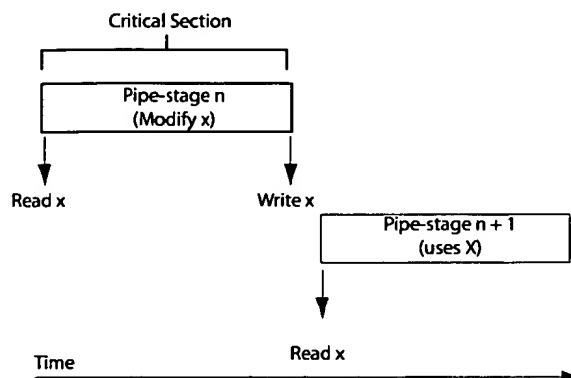
Figure 6: Critical section

Exclusive Modification Privileges between MEs

To ensure exclusive modification privileges between MEs, the following requirements must be met.

- Requirement 1: Only one function modifies the critical section resource.

- Requirement 2: The function that modifies the critical section resource executes in a single pipe stage.

- Requirement 3: The pipeline is designed so that only one ME executes a pipe stage at any one time.
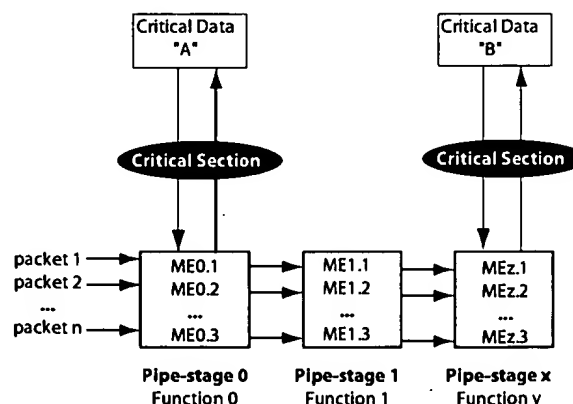

Figure 7: Context pipe with critical sections

Figure 7 shows a context pipeline that supports two critical sections. Each ME is assigned exclusive modification privileges to the critical data, satisfying requirement 1. Requirements 2 and 3 are satisfied because each pipe stage is partitioned into different functions and only one ME executes a specific function.
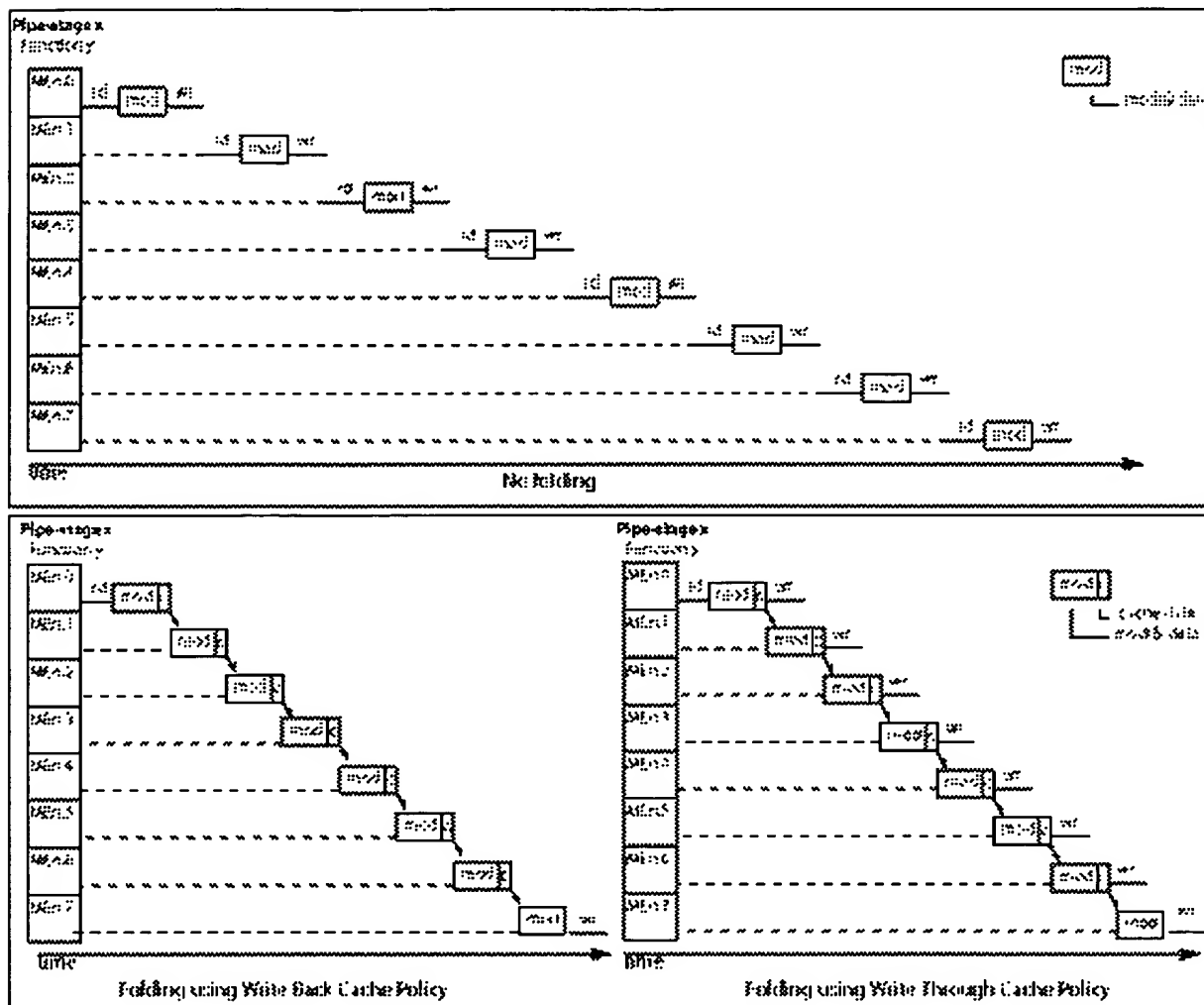
Figure 8: Folding

Folding – Exclusive Modification Privileges between Threads in an ME

Figure 4 also shows a 5 pipe-stage functional pipeline that supports two critical sections. Each pipe stage is assigned exclusive modification privileges to the critical data, satisfying requirement 1. Requirements 2 and 3 are satisfied in the cases shown in the figure; however, it should be noted that a critical section could not be supported by function 1 because it does not satisfy requirements 2 and 3; therefore, two MEs could access the critical data at the same time.

A critical section involves three steps:

1. Reading a resource.

2. Modifying the resource.

3. Writing back the modified data.

As shown in Figure 8, if more that one thread in a pipe stage is required to modify the same critical data, a latency penalty will be incurred if each thread reads the data from external memory, modifies it, and writes the data back. To reduce the latency penalty associated with the read and write, the ME threads can use the MEv2 Content-Addressable-Memory (CAM) to fold these operations into a single read and one or more modifications

A context pipe stage is the only ME that uses the critical data. Therefore the replacement policy for CAM entries is to replace the LRU only on CAM misses. Functional pipelines perform the same function in multiple MEs and therefore are required to evict all the critical data to

external memory before it exits the critical section pipe stage. It should also ensure that the CAM is cleared at the beginning of the pipe stage before any of the threads use the CAM.

Before a thread reads the critical data, it searches the CAM using a critical data identifier such as a memory address. The search will result in one of three possibilities.

1. Miss: When the result is a CAM miss, the critical data is not saved locally, and the thread must read it from external memory. The miss status also includes a Least Recently Used (LRU) CAM entry number.

An ME that executes a context pipe stage will be the only ME that uses the critical data. Therefore, the CAM replacement policy is to replace the LRU only on CAM misses. So the first thing a context pipe stage does on a CAM miss is to evict the LRU data from the local memory back to external memory. Functional pipelines perform the same function in multiple MEs and therefore are required to evict all the critical data to external memory before it exits the critical section pipe stage. Therefore, it can be assured that on CAM miss, the data will already have been evicted. (Note: It is a good programming practice for a functional pipe stage to ensure that the CAM clear instruction is executed at the beginning of the pipe stage before any of the threads use the CAM).

The ME thread then locks the CAM entry and issues a read to get the new critical data. The lock is asserted to indicate to other threads that the data is in the process of being read into local memory. Once the critical data is returned, the thread processes the data, makes any modification to the data, writes the critical data into local memory, and then unlocks the CAM entry.

2. Lock: When the result is a CAM lock, another ME thread is in the process of reading the critical data, and that thread should not attempt to read the data. Instead, it should test the CAM at a later time and use the data when the lock is removed

3. Hit: When the result is a CAM hit, the critical data resides in local memory.

In all cases, the ME thread is assured exclusive access to the data by performing the modification and write operations on the critical data without swapping out.

Pool of Threads

One of the problems with a functional pipeline is that it has a fixed maximum latency, at which time the packet must either be dropped or handed off to a slow path for processing. To get around this, the pool-of-threads concept is proposed.

In place of the functional pipe are 3 blocks:

1. A dispatcher, which assigns packets to available free processing threads.

2. A pool of threads. Each thread makes itself available to the dispatcher for processing tasks. The thread receives a task from the dispatcher, performs the functions of the functional pipeline, then writes results to a re-synchronizing ring.

3. An Asynchronous Insert Synchronous Read (AISR) ring for reordering packets and passing control to the next stage of pipeline. This replaces the elasticity buffer ring.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

INGRESS: IP PACKETS TO CSIX

This section describes the implementation of the 10Gb/sec Packet over SONET design. Table 2 lists seven fundamental tasks performed in the ingress processor. Figure 8 illustrates these tasks mapped to MEs.

Table 2: POS ingress fundamental tasks

| Task	Stage Type	
Frame Assembly Classification	Functional	Performs cell and frame reassembly. Performs IP destination and 5-tuple classification. Maintains the individual data flow contexts. ATM AAL5 CRC checking.
Meter/policing	Context	Single-Rate Tri-color Marker ACLs and flow-based policing.
Congestion management	Context	WRED.
Statistics	Context	Updates statistics counters.
Transmit scheduler	Context	Schedules dequeue for transmit operations. 4-class round robin.
Queue manager	Context	Performs both enqueue and dequeue functions.
Transmit data	Context	Simple transmit to Media Switch Fabric Unit (MSF). buffer management. CSIX segmentation.
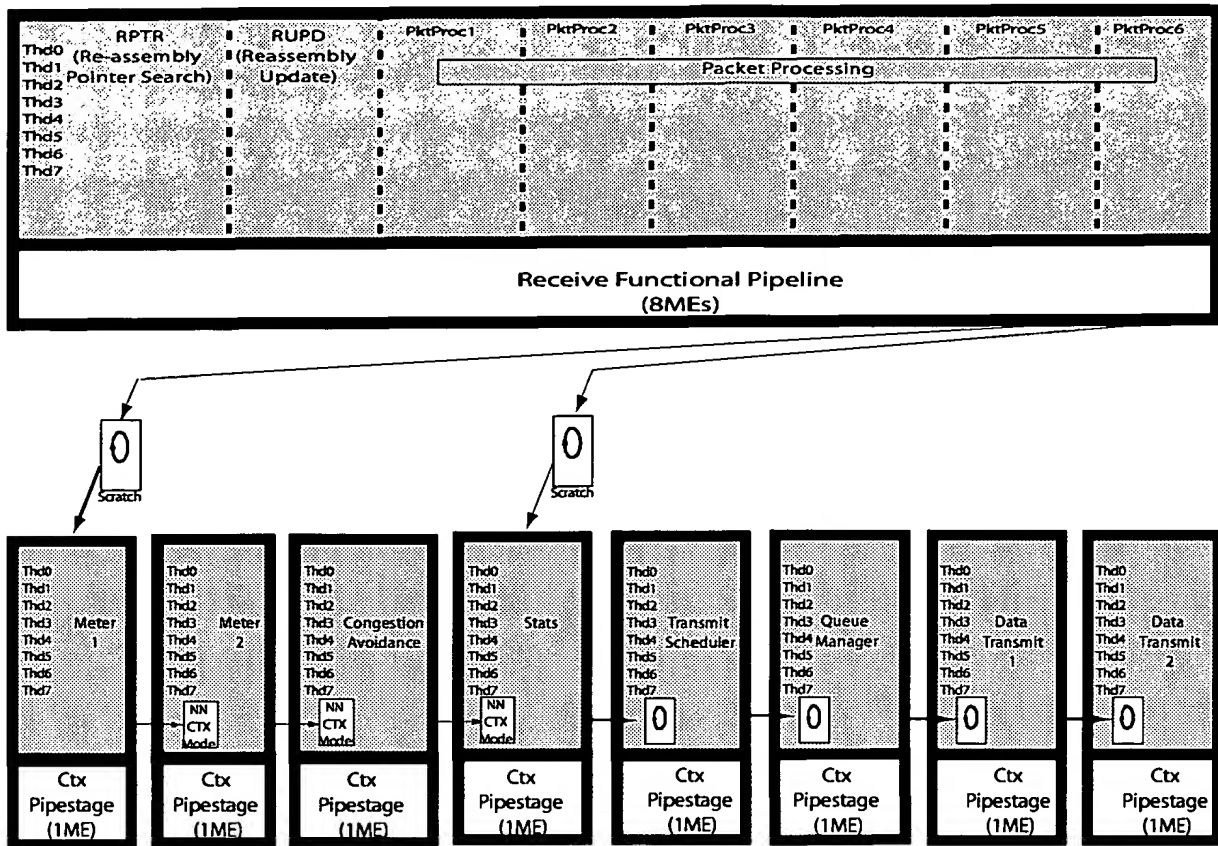
Figure 9: Proof of concept stages

Reassembly Pointer Stage (RPTR)

The RPTR pipe stage finds the pointer to the reassembly state information in SRAM so that the Reassembly Update (RUPD) critical section can perform its read-modify-write on the data as fast as possible. The SPI-4 and CSIX interfaces segment packets into smaller SPI-4 and CSIX frames. The RPTR, RUPD, and packet processing pipe stages work together to reassemble these segmented frames back into full packets. The reassembly state information is used to keep track of the reassembly process.

The RPTR pipe stage also looks at the L2 header, to determine the offset into the IP header. This offset is used by the next pipe stage (RUPD) during reassembly. It is not included in the RUPD because the RUPD is a critical section.

Reassembly State Update Stage (RUPD)

RUPD pipe stage is a critical section that provides the packet processing pipe stage with a pointer to the location in DRAM where the network data should be assembled. It is a critical section because it modifies the

reassembly state information that is maintained in SRAM. The RUPD pipe stage manages this data structure exclusively. Managing the reassembly state involves allocating buffers and calculating offsets, byte counts, and other variables. The MEv2 CAM is used to maintain a coherency of the reassembly state between the eight threads. If is important to note that if the RBUF Control indicates that the complete packet is in the RBUF element, the reassembly information does not need to be accessed and updated. This greatly reduces the memory system bandwidth for packets of 128 bytes or less.

Packet Processing (PPR)

The packet processing pipe-stage threads complete the reassembly process by writing the data to the buffer and also looking at the L2 through L7 packet headers to process the packet. This pipe stage is very application dependent and is expected to change from one application to another. The initial reference design will support the following:

- Stripping the PPP header from the packet.

- Verifying the IP header.

- Determining the destination port with an IP destination search.

- Identifying flows and support access lists with a 7-tuple search.

- Modifying the IP header TTL and checksum.

- Appending a new level-2 header for the packet, containing the classification state to be used by the egress processor.

The packet processing pipe stage is *not* a critical section. It executes six times longer than the other pipe stages in the functional pipeline. The packet processing pipe stage ends with a synch section to ensure that packet order is maintained.

Metering 1 and Metering 2

The Single-Rate Three-Color Marker (srTCM) meters an IP packet stream and marks its packets either green, yellow, or red. Marking is based on a Committed Information Rate (CIR) and two associated burst sizes: a Committed Burst Size (CBS) and an Excess Burst Size (EBS). A packet is marked green if it doesn't exceed the CBS, yellow if it does exceed the CBS but not the EBS, and red otherwise [2].

Metering is performed when an EOP is received for a packet. The packet processing pipe stage performs a flow identification search and gets a pointer to the metering information. This pointer is passed to the metering functions. Metering is performed in two pipe stages (Meter 1 and Meter 2). The first stage reads the pointer from a scratch ring, reads the metering parameters, and calculates the number of tokens collected. The second metering pipe stage performs the actual metering function and writes the updated meter information back to memory. Note that the two pipe stages access and use the same metering parameters. The MEv2 CAM in both pipe stages is used to maintain coherency of the parameters between the threads with an ME and between the two pipe stages. This is possible because the ME threads process packets in strict order and any CAM hit in the first pipe stage is guaranteed to be in the CAM of the second pipe stage. Note that the reverse is not true since the first pipe stage will be working on eight new packets while the second stage is processing its pipe stage.

Congestion Avoidance

Congestion avoidance techniques monitor network traffic loads in an effort to anticipate and avoid congestion at common network bottlenecks [1]. One of

the ways in IP QoS is through packet dropping[1]. This requires a way to estimate the average queue size on each packet arrival. Basically, after the packet is classified and before it is enqueued, this block looks at the average size of the queue, compares it to certain thresholds, and makes a decision on accepting or dropping the packet.

RED

The Random Early Detection (RED) was originated from S. Floyd and V. Jacobson [3]. According to their paper, the gateway detects incipient congestion by computing the average queue size. The gateway could notify connections of congestion either by dropping packets arriving at the gateway or by setting a bit in packet headers. When the average queue size exceeds a preset threshold, the gateway drops or *marks* each arriving packet with a certain probability, where the exact probability is a function of the average queue size.

RED gateways keep the average queue size low, while allowing occasional bursts of packets in the queue. During congestion, the probability that the gateway notifies a particular connection to reduce its window is roughly proportional to that connection's share of the bandwidth through the gateway. RED gateways are designed to accompany a transport-layer congestion control protocol such as TCP. The RED gateway has no bias against bursty traffic and avoids the global synchronization of many connections, decreasing the connections' window at the same time.

WRED

WRED combines the capabilities of the RED algorithm with flow-specific DSCP and associated RED parameters. This combination provides for preferential traffic handling for higher-priority packets. It can selectively discard lower-priority traffic when the interface starts to get congested and provide differentiated performance characteristics for different classes of service.

Statistics

The ingress processor supports statistics for incoming traffic, while the egress processor supports statistics for traffic from the switch fabric. Statistics for the flow based on the 7-tuple lookup are also kept for packets_transmitted and packets_dropped. This section applies to both the ingress and egress processors.

---

[1] Floyd and Jacobson [3] suggest marking instead of dropping. The congestion and ECN-capable bit are only recently being defined in RFC 2481 as an experimental protocol.

RFC2863 [4] states that 64-bit counters must be supported for interfaces that operate at data rates greater than 650Mbps; otherwise, the rollover rate will be too frequent for a monitor program to maintain. Although the standard states that 64-bit statistics are required, it is not absolutely necessary since the rollover rate is 468 years. The POS Proof of Concept implementation uses 63-bit statistics that roll over once every 234 years.

The POS Proof of Concept design includes various assumptions concerning statistics:

1. There are two separate types of statistics maintained for data arriving from the media interface: those associated with an IP interface and those associated with a flow.

2. The IP interface statistics are limited to one set per physical port.

3. Each of the flow and IP interface statistics requires one byte count, one packet count, and one timestamp indicating the last time something was received.

4. Both the flow and IP interface statistics may require 63-bit statistics.

Transmit Scheduler

The transmit pipeline begins with the transmit (Tx) scheduler context pipe stage. The Tx scheduler schedules packets to be transmitted to the CSIX fabric and passes an enqueue and dequeue request onto the next pipe stage (queue manager). The transmit scheduler is also responsible for maintaining the queue status state.

The scheduling algorithm is implemented in software and can be modified per customer requirements. The assumption made for the example design is that the scheduler is transmitting into a switch fabric that supports 16 line cards, each line card having eight ports, and that each port on a line card supports four class types. The total number of ports in the system is 16 x 8 or 128. The example design implements a hierarchical scheduling algorithm. There are four class (priority) wheels, each with 128 entries. The 128 entries each represent a port. Each class wheel is serviced with a Round Robin (RR) scheduling algorithm. A programmable distribution wheel is used to select which ring gets service at each scheduling interval. The programmable distribution wheel provides an anti-starvation mechanism that ensures fairness with some degree of programmability. The example design uses a work-conserving algorithm (it searches for work to do during each scheduling time). If no work is found on a particular class wheel, then the search moves to the next wheel entry.

The transmit scheduler is made up of two threads: schedule (Thread 0) and flow control (Thread 2). Note that the scheduler ME is configured to operate in 4 context mode; therefore, the four threads are numbered 0,2,4,6. Threads 4 and 6 of the scheduler ME are unused.

The scheduler thread uses a singly linked list in local memory (LM) to determine the next eligible schedule for each class. Each class wheel has a 32-bit control register that is stored in LM. This control register is made up of a 16-bit previous queue pointer and a 16-bit current queue pointer that maintain the active links for the class wheel. Each link list entry is made up of an 8-bit next queue pointer, 23-bit queue cell count, and 1-bit flow control status that is stored in LM.

Enqueue information is passed from the receive pipeline through the statistics ME to the scheduler ME. The statistics ME computes the number of cells that each enqueue packet contains from its enqueue state. The statistics ME encodes the computed cell count in one of four enqueue words that are passed to the scheduler ME via a Next Neighbor (NN) FIFO. The scheduler ME uses the cell count information to maintain the queue status for each queue.

Queue Manager

The queue manager is responsible for performing enqueue and dequeue operations on the transmit queues for all packets. Although IXP2800 can support either a linked list or ring queue structures, this implementation of the queue manager is designed to support the linked list queue structure.

Transmit 1 and Transmit 2

The transmit data pipe stages receive a transmit request from the queue manager and, in response, segment the buffer packet data into c-frame segments and moves the data into a TBUF so that the MSF transmit state machine can send the data to the fabric. When all the data in a buffer is transmitted, it frees the buffer by placing it onto a buffer free list.

The queue manager places transmit requests onto a next-neighbor ring. The transmit data pipe stage reads the request and processes the request.

To process the request, the transmit data thread must first check the CAM to see if the buffer descriptor is cached in local memory. If not, it evicts the LRU buffer descriptor to SRAM and reads in the new buffer descriptor. When the buffer descriptor is local, the transmit data thread checks to see if it is an SOP. If so, it finds the current packet data location and offset into the buffer and reads the IP header into transfer registers,

updates the IP TOS field with the DSCP, and updates the IP header checksum. Then it submits the data to the MSF for transmit, with the following steps:

1. Writes the IP header and the CSIX L2 A and B headers to the TBUF element data.

2. Moves the remaining data from the DRAM to the TBUF.

3. Validates the TBUF element so that the MSF transmit state machine knows to send the data to the fabric.

If the request is not an SOP, the transmit data thread finds the current packet data location and offset into the buffer and moves all the data from the DRAM to the TBUF and validates the TBUF element. If the request is not an EOP, it places the buffer address onto a buffer free list after the entire packet has been transmitted.

CONCLUSION

Performance was measured for a variety of packet streams, notably single OC-192 flow, 16 flow, and >16 flow cases for minimum-sized packets, and a distribution of packet sizes for non-minimum-sized packets. The context pipe stages tend to govern performance that is a steady rate of one packet per 55.6 ME cycles.

The elasticity buffers need to accumulate messages before the following context pipes could achieve a full rate without getting empty ring messages.

The 10 Gigabit/sec performance is achievable with careful attention to the instruction counts for context pipe stages, and the overall latency/number of threads for the functional pipe stages.

ACKNOWLEDGMENTS

The authors acknowledge Gil Wolrich, Hugh Wilkinson, Deb Bernstein, Michael Fallon, Sanjeev Jain, Stepanie Hirnak, David Romano, Mark Rosenbluth, and John Wishnewski.

REFERENCES

[1] Matthew Adiletta, et al., "The Next-Generation Family of Intel Network Processors," *Intel Technology Journal*, Q3, 2002, V6 Issue 3.

[2] Internet Network Working Group, RFC 2697, September 1999.

[3] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, V.1 N.4, August 1993, pp. 397-413.

[4] Internet Network Working Group, RFC 2863, April 1998.

AUTHORS' BIOGRAPHIES

Matthew Adiletta is an Intel Fellow and Director of Communication Processor Architecture. He led the architectural development and implementation of the IXP2800 and is driving the IXP roadmap. He is interested in processor architecture and advanced implementation techniques for rapid silicon development. He is also intrigued with the semantic web and network security. Adiletta received his B.S. degree in Electrical Engineering, with Honors, at the University of Connecticut. He resides in Bolton, Massachusetts. His e-mail is matthew.Adiletta@intel.com.

Donald Hooper is a senior software architect in the Network Processor Group. He has led many projects including Logic Synthesis, Video Servers, MPEG 2 and DAVIC Standards, IXP1200 Software Tools and Libraries, IXP2800 Proof of Concept Designs, and NPG Coding Standards. His professional interests include networking, artificial intelligence, and object-oriented languages. He attended four colleges with cumulative B.S.E.E. degree credits finishing at UCLA. He resides in Shrewsbury, Massachusetts. His e-mail is donald.hooper@intel.com.

Security: Adding Protection to the Network via the Network Processor

Wajdi Feghali, Brad Burres, Gilbert Wolrich, Douglas Carrigan
Intel Communications Group, Intel Corporation

ABSTRACT

As information security becomes more important in today's world, it is necessary for networking equipment to enable cryptographic functions. The Intel IXP28xx network processor integrates the appropriate set of features to enable the addition of security functionality with significant advantages over a discrete solution.

The integration of security functionality in the IXP28xx network processor was initiated by the investigation of cryptographic systems requirements. Several possible architectures were investigated, but they all lacked certain key elements. It was proposed that an integrated solution capable of network processing and security processing offered the most advantages.

After determining that an integrated solution had the most advantages, we found it necessary to understand what new functions needed to be integrated and what existing IXP2800 functions could be leveraged. The new functions were identified, implemented, and interfaced to the rest of the network processor. These new functions, combined with the baseline IXP28xx, make it possible to provide secure traffic at 10Gigabits/second.

INTRODUCTION

With the increasing reliance of businesses and individuals on computer networks, there is a corresponding increase in the importance of information security. This increase requires that some base security functionality, such as data confidentiality and data integrity, be afforded to every packet placed on the network. Despite this requirement, security is sometimes an after-thought in many networking equipment designs. It is essential to provide security functionality as part of the networking building blocks from the beginning of the design cycle.

Previous designs have added security to the network through either a co-processor or an inline security processor. As data rates go up, the co-processor solution becomes less and less practical. Inline security processors can actually scale to the higher data rates but must perform many of the same functions as the network processor does to achieve the high data rate.

Integrating security functions onto the IXP28xx makes it possible to provide secure network traffic at 10Gigabits /second while using the same system designs as for an ordinary network processor. This enables the design of security functionality in network equipment from the start and at a lower overall system cost in terms of power consumption, board real estate, and silicon investment.

This paper describes how security functionality is added to the IXP2800 network processor in order to support data confidentiality and data integrity. Specifically, it discusses the hardware features added and how these features can work in cooperation with the rest of the network processor functionality.

SECURITY SYSTEMS ARCHITECTURES

There are three primary ways to add security functions to networking hardware equipment.

The first and most common method today is to use a co-processor coupled with a network processor or a general processor. As data rates go up, this method becomes less practical because the packet must traverse shared resources such as data buses or memory four times.

The second method is to add a security processor inline with a network processor. While this approach can achieve high data rates, the inline security processor must perform many of the same functions that the network processor must do such as packet reassembly; thus work must be repeated and silicon area must be duplicated.

The third method is to add the security functionality into the same silicon as the network processor, thus adding security functionality into the network processor while maintaining wire rate and minimizing new silicon area. As

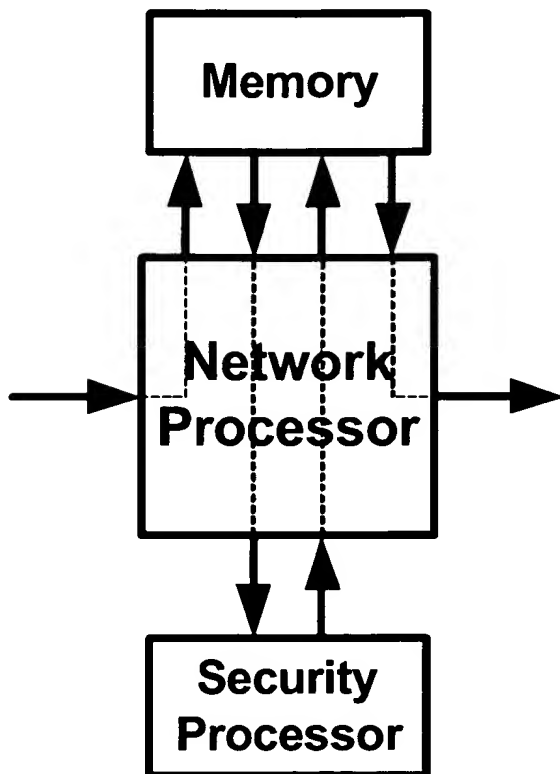new network line cards are designed, an integrated solution will prove beneficial.

Figure 1: Look-aside architecture

assemble the packet in memory before sending it to the security processor and that the packet is then placed in memory before being transmitted. Although the bus between the network processor and the security processor must be able to handle twice the line data rate, requiring many more pins on the network processor, the network processor memory must be traversed four times, and that is not desirable for high packet rates. It is important that the bulk of the data traverse the memory just twice: once for write and once for read.

Figure 1 shows a network processor and a security processor in the look-aside configuration. The packet data must typically traverse the memory four times, as opposed to two times, and the bus between the network processor and security processor must be capable of doing at least twice the desired wire rate.

Flow-through Architecture

The flow-through architecture solves the performance problems of the look-aside architecture, but it requires the security processor to do many of the functions that the network processor is targeted for. Some of these tasks include reassembly of packets, protocol processing, and exception handling.

It is very desirable to be able to use the same underlying hardware architecture to target multiple applications. This would be possible if the security functionality were added to the network processor.

Figure 2 shows partitioning of flow-through architectures. Some applications require that the network processor be placed before the security processor, some require that the security processor be placed before the network
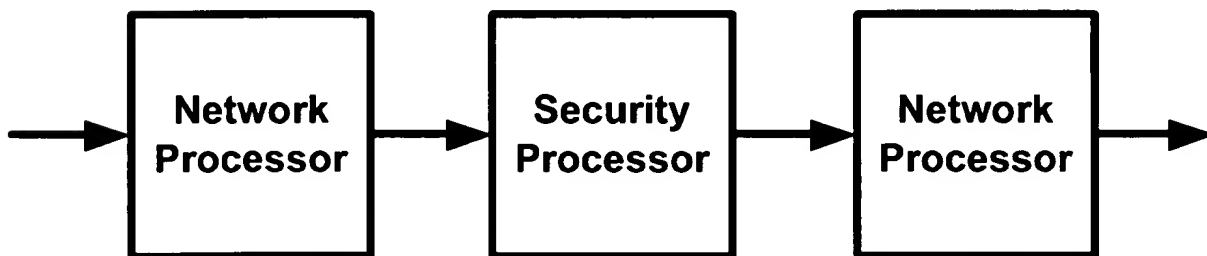

Figure 2: Flow-through architecture

Look-aside Architecture

The look-aside architecture is attractive because it places fewer burdens on the security processor to completely implement protocol processing. For a simpler security processor design, a greater burden is placed on the network processor. Current security processors require that the entire packet be available before processing can begin. This implies that the network processor must

processor, and some require a network processor before and after the security processor. For example, an SSL proxy application requires termination of a TCP connection, SSL processing, and then the establishment of a new TCP connection.

Security: Adding Protection to the Network via the Network Processor

Protocol and Cryptographic Processing

The securing of network traffic can essentially be portioned into two partitions: protocol processing and cryptographic algorithm processing.

Protocol processing would include Encapsulating Security Protocol (ESP), Authentication Header (AH), Secure Sockets Layer (SSL), Transport Layer Security (TLS), and other non-security protocols such as Transmission Control Protocol (TCP) and the Internet Protocol (IP) processing.

Cryptographic algorithm processing includes the data manipulation that would need to be done on the entirety of the payloads, such as confidentiality and integrity.

The flow-through architecture requires the security processor to do similar functions as the network processor, mainly protocol processing. This can be done with dedicated hardware or perhaps one or many protocol processors integrated in the security processor.

An interesting question is raised: Is it better to add the necessary functions to a network processor to enable it to target security processing, or add network processor functions to the security processor to enable it to completely handle protocol processing?

We decided to leverage the extensive functionality and flexibility of the IXP2800 network processor and add to it the necessary cryptographic algorithms.

LEVERAGING THE IXP2800

The IXP2800 consists of several units that are connected via the chassis. The chassis connects the following units that can be directly used for security processing: microengines, SRAM, DRAM, lookup hash, PCI, and XScale™, via a set of command buses and data buses. The cryptography units are added to the chassis and are accessible by the microengines and the media switch fabric interface for data, and by the microengines for commands. For details on the chassis, see reference [7].

Microengines

Microengines are used to do the protocol processing, such as ESP processing for IPSec traffic. That processing includes constructing new headers, copying fields from one header to another, and modifying security state information. The microengines are designed for packet processing. For example, during replay checks on incoming ESP packets, it is necessary to read the sequence number state, modify the replay window, and write the data back to memory.

The microengine CAM is used to effectively manage the reading, make multiple modifications across several microengine threads, and write back the sequence number state to memory. This technique is called *folding* [6].

DRAM

The IXP2800 provides three independent channels of DRAM memory, yielding enough capacity to handle millions of security associations and enough throughput to handle 10Gigabit/second IPSec wire rates.

Hash for Lookups

Hashing for lookups can be used to find the required security association information for a given packet. Although other methods can be applied, combining a dedicated lookup hash unit with the use of external SRAM yields a cost-effective mechanism to conduct many required lookups.

SRAM

An important aspect of security processing is to find security associations in order to apply the appropriate cryptographic algorithms for a given flow. The four SRAM channels that the IXP2800 supports can be used to store hash tables.

PCI

When establishing security association, some operations, such as public key computations, are required. A co-processor that is connected to the PCI bus can conduct these operations.

XScale

The IXP2800 includes an XScale processor that executes general-purpose code. The XScale processor can be used for exception handling for packet processing or session setup protocols such as the Internet Key Exchange (IKE) protocol.

IXP2800 Feature Benefits

The IXP2800 allows several feature benefits for security processing. These benefits include flexibility of the implementation of the protocols, optimization of the protocols for certain applications, and the implementation of different applications using the same underlying hardware.

Protocol Flexibility

The security functionality is designed to allow support for many protocols, such as IPSec, SSL/TLS, ATM, and future protocols that use 3DES, AES, and SHA-1.

It has been proposed that the ESP protocol support larger sequence numbers. This can be achieved when combining the flexibility of the microengines and the flexibility of the cryptography unit. It is also possible to support IPv4 and IPv6 while using the underlying cryptographic hardware.

This flexibility is achieved by relying on the microengines to provide the packet processing such as header manipulation policy enforcement and post decryption payload inspection.

Protocol Optimizations

It is also possible to make optimizations depending on the application being targeted. For example, when optimizing for storage applications, large packets and few sessions are required. This makes it possible to use the microengines for adding more features. The flexibility of the network processor is key for enabling new applications because it is possible to allocate the available resources, given the new application requirements.

leverage, it definitely provided some design challenges. Since the security functions are somewhat orthogonal—depending on the configuration—it is desirable to fully utilize all security hardware in parallel. Enabling this parallelism in hardware requires careful management of common components such as global buses, local memory, and data stalling methods. It also requires substantial dexterity in switching the IVs, keys, and other state information when switching packet flows, without sacrificing performance.
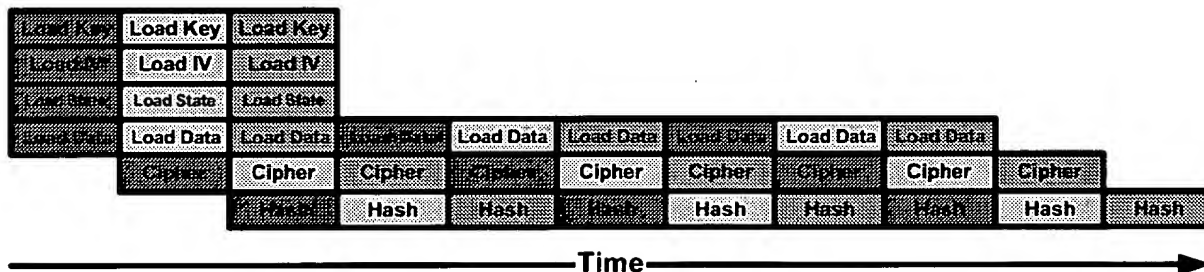
Figure 3: Security processing, pipelining, and interleaving using three wires and one core

Merging Packet Processing and Security

When designing a network security product, one must consider both the packet-processing requirements and the security requirements. A general-purpose processor coupled with a security co-processor will not be fast enough to achieve 10Gigabit/second rates with existing products. It is possible to couple an existing network processor, such as the Intel IXP1200, with a security co-processor, but today's security ICs offer only co-processor architectures, and these are insufficient.

Pipelining Security Processing

It is important to add the cryptographic functionality in such a way as to leverage the network processor features, such as multiple threads per microengine and multiple microengines.

Figure 3 shows pipelining of the functions that need to be executed to encrypt and hash data. Each color represents a different packet. The cipher key, IV, SHA-1 state and data loading can be viewed as one stage of the pipeline. While the red packet segment is being encrypted, new state is loaded for the yellow packet. While the red packet segment is being hashed, the yellow segment is being encrypted, and new state is being loaded for the green packet. For longer packets, no new state needs to be loaded, and only the data manipulation is pipelined.

Each microengine can have up to eight threads of execution running. While this multi-threaded model is one of the strengths of the architecture that we chose to

It is also important to pipeline the protocol processing. For example, when processing IPSec tunnel mode packets, it is possible to pipeline all the required processing.

Aggregating without Reassembly

Figure 4 shows a 3DES core with access to three IVs and three cipher keys. It is possible to choose which IV and key pair is used on a block-per-block basis without incurring any overhead cycles.
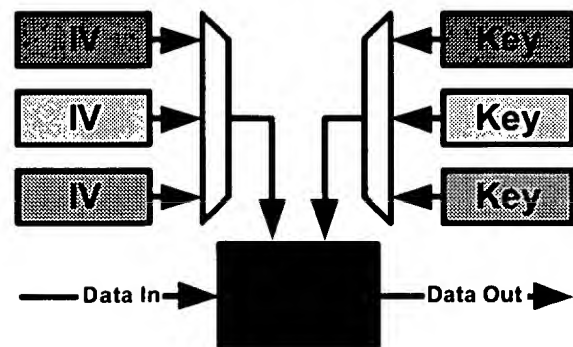
Figure 4: Multiple keys and IVs

Although it is important to achieve 10Gigabits/second rates on a single interface, it is also important to aggregate, for example, ten 1Gigabit/second interfaces. When multiple interfaces are connected to the network processor, the data of a particular packet might be interleaved with other packet data in the receive buffer. Typically, the network processor reassembles the packet

in memory. If encryption needs to be performed on the packet, then the packet must necessarily traverse the memory four times before it is sent off-chip.
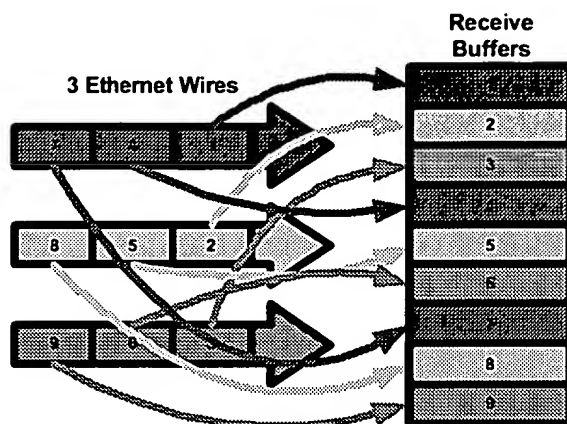

Figure 5: Multiple wire aggregation

Figure 5 shows aggregation of three wires. The packets on each wire are contiguous, but they are presented to the IXP2800 network processor interleaved with packet segments of other wires.

To avoid the reassembly before encryption, the cryptography unit supports enciphering several packets while maintaining state, such as cipher keys, cipher IVs and SHA-1 state, internally to the cryptography unit. This avoids having to reassemble the packet in memory and hence saves memory bandwidth. The states can be swapped on each block of an algorithm without any overhead cycles.

The IXP2800 supports software techniques to handle reassembly of segmented packets. The techniques are divided into two stages: Reassembly Pointer Stage (RPTR) and Reassembly State Update (RUPD). More information on RPTR and RUPD techniques can be found in [6].

Post-Decryption Processing

Given that once a packet is decrypted, further processing is required. To reduce latency, it is important that the post-encryption processing can be started before the packet is fully decrypted. This is very important for larger packets.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

CRYPTOGRAPHY UNIT MICRO-ARCHITECTURE

The cryptography unit is comprised of several algorithms that in conjunction provide data confidentiality and data integrity. Each algorithm has its own set of trade-offs and challenges, in terms of silicon area, parallelism, and symmetry.

The added security functionality supports the Data Encryption Standard (DES), 3DES, and the Advanced Encryption Standard (AES) algorithms along with the Secure Hash Algorithm (SHA-1) for data authentication directly in hardware.

Figure 6 shows the data path of a cryptography unit. It consists of two 3DES cores, one AES core, and two SHA-1 cores. It is possible to process the data via the SHA-1 cores either before or after the ciphers have processed the data. The IXP2800 has two such cores.

Pipelining Around Cipher Block Chaining

During Cipher Block Chaining (CBC), encryption pipelining across several blocks of the same packet is not possible. This is because the results of the previous cipher operation are used in the calculation of the next block of data.
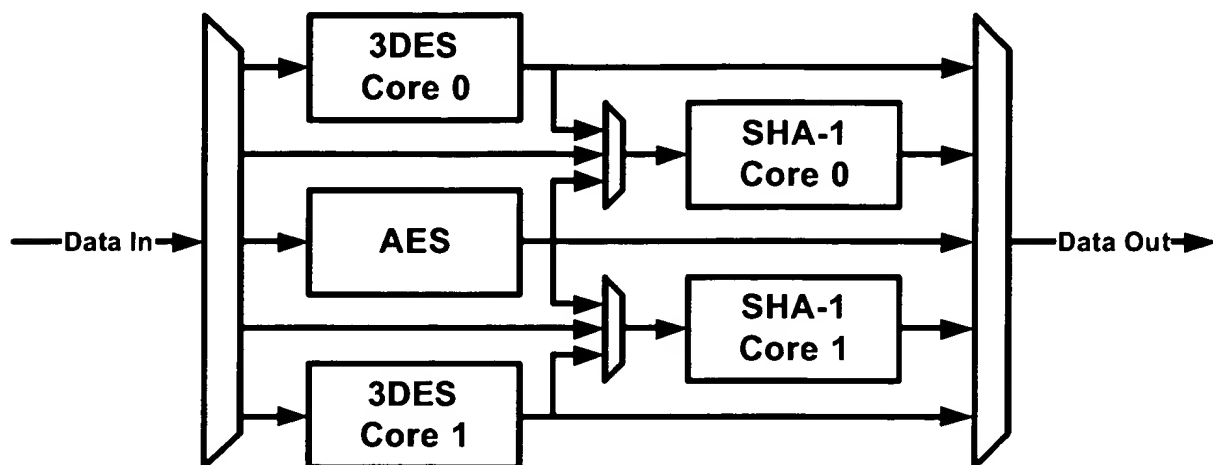
Figure 6: Cryptography unit overview

Figure 7 shows the dependency in cipher block chaining that requires the cipher text to be XORed with the next clear text block. This makes it very desirable to provide a low latency cipher algorithm implementation.

By providing each core with access to multiple keys and IVs with no overhead on transitions and by using more than one core, it is possible to overcome the CBC problem.
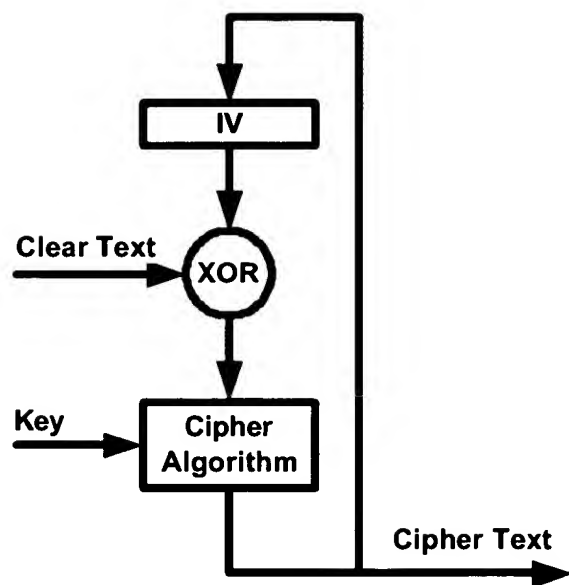

Figure 7: Cipher block chaining critical path

Although newer modes of operation do not have such a direct dependency between clear text and cipher text, it is important to support legacy modes.

Data Encryption Standard

This cipher block algorithm is a straightforward design. Since 3DES requires the performance of three DES rounds with different keys in varying modes, the hardware grabs all required keys and uses the same DES round hardware three times. The hardest design challenge was the SBOX implementation. The SBOX is a 64-entry lookup that is arrayed out 24 times and is used three times serially. In order to optimize the performance, a custom designed lookup table implemented as a passgate mux structure was used.

Advanced Encryption Standard

The AES algorithm represents unique challenges: 1) it is designed to favor encryption speed; 2) the encryption and decryption, at a first glance, require different circuits; and 3) the AES algorithm is the support for multiple key lengths.

AES is implemented both to give the same speed between encryption and decryption and to use the same circuit for all key sizes. This was challenging from an analysis and implementation point of view.

AES Key Scheduler

The key scheduler needs to support both encryption and decryption and all three key sizes. Figure 8 shows the key scheduler circuit. It is possible to load either a 128-bit, 192-bit, or 256-bit key in the available 32-bit registers. Once the key is loaded, it is possible to output the round keys for both encryption and decryption. For decryption it is necessary to compute the encryption key schedule at setup time and then load the end of the encryption key schedule for decryption processing. The same hardware is

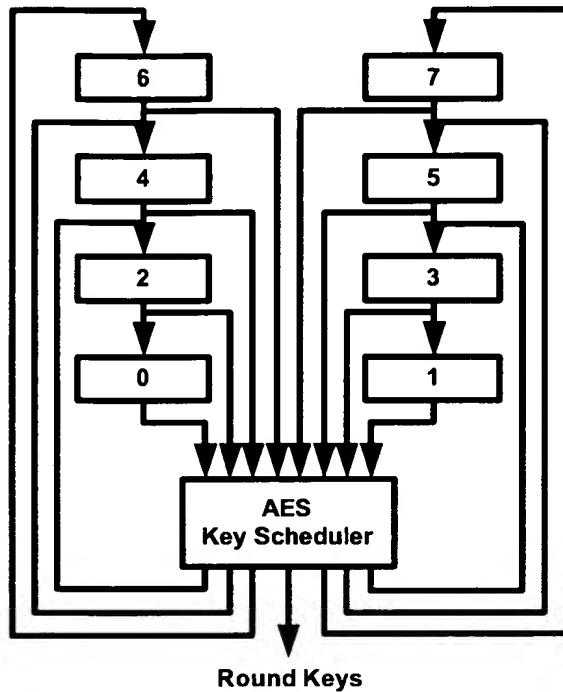capable of doing both the encryption and the decryption key schedule on the fly.

Round Keys

Figure 8: AES key scheduler

Secure Hash Algorithm and HMAC

The secure hash algorithm is used for data authentication. Depending on the current protocol, SHA-1 can operate either on the unmodified packet data or on the packet data after it has been modified by one of the cipher algorithms. It operates on a 512-bit block size and requires a data buffer to accumulate ciphered data. The data buffer design is important because it breaks the dependency between the data source and the actual hashing algorithm. By breaking this dependency, it creates a less complex design that enables the cipher algorithms and the SHA-1 algorithm to run at different rates.

The secure hash algorithm is used in the HMAC protocol and adds significant overhead on short packets. For example, processing a 40-byte IP packet with IPSec requires that 3 hashes be processed, assuming that the inner pad and outer pad XORed with the key are precomputed.
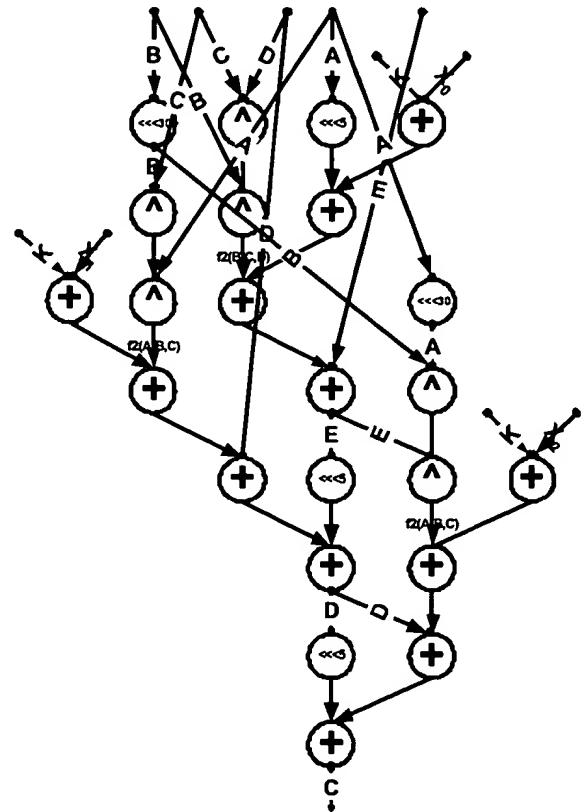

Figure 9: SHA-1 critical path

Figure 9 shows the start of the SHA-1 critical path analysis. This was done to exploit as much parallelism from the algorithm as possible while minimizing the required silicon area. The critical path analysis is then converted into a circuit.

Verification

Verification of the cryptography unit was conducted using multiple approaches. Given the amount of parallelism, it was important to validate functionality with different instruction sequences, and it was also important to validate the algorithms independently.

Cycle-Accurate Model

A C++ cycle accurate model was hand-written to provide a concrete model and a method for comparing the AES key scheduler against known results and for providing intermediate data for circuit debugging. The C++ cycle-accurate model also provided an initial overview of the control logic that would be required.

This model proved to be very successful in allowing us to try out new ideas in a short time and to validate the circuit implementation.

Instruction-Accurate Model

An important aspect of the cryptography unit architecture was the amount of parallelism that was provided. Although the individual algorithms can be verified with the cycle-accurate model, it was also important to verify that a sequence of commands produced the appropriate results. The instruction-accurate model served that purpose. Given the initial state of the cryptography unit, a sequence of commands, and the final state of the cryptography unit, it was possible to verify that the appropriate steps were carried through appropriately.

PERFORMANCE

The cryptography unit is designed to achieve 10Gigabits/second Ethernet performance when using IPSec. Although packet rate performance is important, consideration must be given as to the overall power consumption and the key agility of the system.

IPSec Performance

The cipher data path delivers over 25 million IPSec packets per second with a 40-byte clear text payload.

The SHA-1 data path delivers over 10 million HMACs per second for 40-byte clear text payloads.

This is sufficient performance to encrypt and authenticate IPSec at 10Gigabit/second Ethernet rates when 100% of the traffic needs to be secured.

Figure 10 shows the IPSec performance for a 10 Gigabit/second Ethernet wire. When securing packets with IPSec tunnel mode, the original packet is encapsulated in a new packet, and the secure packet grows in size. The graph shows the IPSec and clear packet rate in Gigabits/second and the IPSec packet rate in millions of packets/second.

Power Consumption

One big benefit of adding security functions to the IXP2800 is a dramatic savings in power consumption for the overall solution. If current security processors were scaled to 10Gigabits/second rates, they would require between 13 and 30 Watts. Between the meticulous attention given to reducing power consumption during the design phase and, more importantly, the lack of I/O devices and overlapping silicon functionality, the IXP2800 with the Cryptography Unit reduces power by a 10X factor over its competitors. This is a tremendous benefit given the tight power constraints facing many networking equipment manufactures.

Key Agility

Key agility is important when considering the small packet rates of 10 Gigabit/second networks. The cryptography unit allows loading keys while the cryptographic algorithms are running. The cipher algorithm key schedulers run independently from the ciphers data circuits. This enables the encryption of all traffic, even minimum packet sizes, without sacrificing performance.
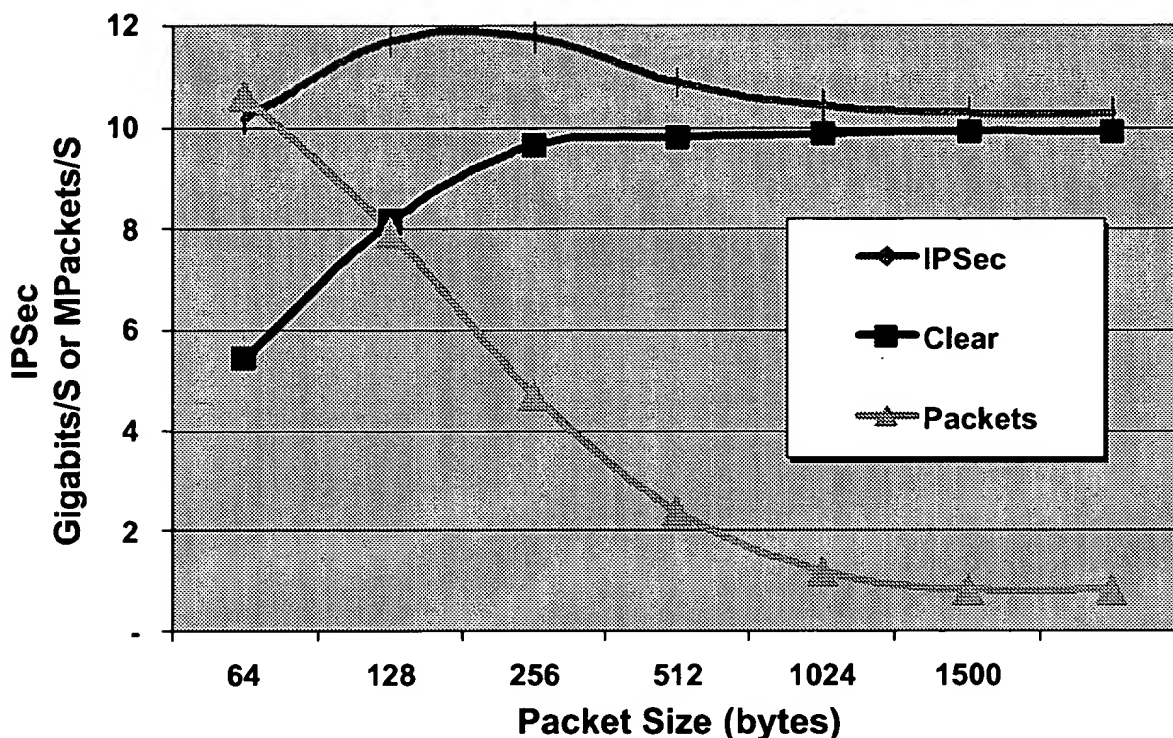
Figure 10: Performance estimates of the cryptography units doing IPSec ESP 3DES SHA-1 for 10Gigabits/second

CONCLUSIONS

The problem of providing a secure network is one that networking equipment manufactures must tackle. The integration of security functionality onto the IXP28xx enables these manufacturers to easily solve the problem of providing that security.

The cryptographic algorithms that are implemented in silicon work closely with the other IXP28xx silicon and software resources to achieve secure data rates of up to 10Gigabits/second. Achieving the security processing requirements on a network processor makes it possible to provide not only a high-speed secure connection but also one that is achievable at a relatively low cost.

Due to the flexible nature of the IXP28xx network processor, future generations will integrate more security functionality. These functions might include public key acceleration, random number generation, and intrusion detection.

By providing both the general-purpose network processor and the security processing integrated onto a single silicon die, Intel is working to secure its position as the leading silicon provider for the networking equipment manufacturers.

ACKNOWLEDGMENTS

The authors acknowledge the contributions of John Cyr and Matthew Adiletta.

REFERENCES

[1] Federal Information Processing Standards Publication 180-1, Secure Hash Standard, May 11, 1993.

[2] Federal Information Processing Standards Publication 46-3, Data Encryption Standard, October 25, 1999.

[3] Federal Information Processing Standards Publication 197, Advanced Encryption Standard, November 26, 2001.

[4] A. Menezes, P. van Oorshot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[5] W. Feghali and B. Burres, IXP2000 Cryptography Unit, Engineering and Architecture Specification, April 30, 2002, Intel internal document.

[6] "Packet over SONET: Achieving 10 Gigabit/sec Packet Processing with an IXP2800," *Intel Technology Journal*, Vol. 6 issue 3, August 2002.

[7] M. Adiletta, et. al,"The Next Generation of Intel IXP
Network Processors," *Intel Technology Journal*, Vol. 6
issue 3, August 2002.

AUTHORS' BIOGRAPHIES

Wajdi Feghali is a security architect in the Network
Processor Group. He has been with Intel for two years
leading the IXP2800 hardware and software security
architecture. Prior to Intel, Wajdi worked for TimeStep and
then NewBridge Corporations, developing secure VPN
gateways. Wajdi attended the University of Ottawa and
obtained a degree in mathematics in 1997. He resides in
Ottawa, Ontario, and can be reached via e-mail at the at
wajdi.k.feghali@intel.com.

Brad Burres is a senior component design engineer in the
Network Processor Group in Hudson. He is a five-year
veteran at Intel, having spent four of those years working
on network processors, including the IXP12xx and IXP28xx
families. Brad did the micro-architecture and ASIC
implementation for the IXP28xx security solution. Brad
received a B.S. degree in Computer Engineering from the
University of Arizona. He resides in Cambridge,
Massachusetts, and can be reached via e-mail at
brad.a.burres@intel.com

Gilbert Wolrich is a senior architect in the Network
Processor Group in Hudson. He has contributed to the
definition of both the IXP1200 and IXP2000 solutions. Gil
has worked on high-performance network and general-
purpose processors, and numerous floating-point units,
and is interested in network security. Gil received a B.S.
degree from R.P.I. and an M.S. from Northeastern
University in electrical engineering. He resides in
Framingham, Massachusetts, and can be reached via e-
mail at gilbert.wolrich@intel.com.

Douglas Carrigan is a strategic marketing manager in the
Network Processing Group in Hudson, Massachusetts.
His e-mail address is douglas.carrigan@intel.com.

IXA Portability Framework: Preserving Software Investment in Network Processor Applications

Uday Naik, Alex Shoykhet, Larry Huston, Donald Hooper, Raj Yavatkar, Duke Tallam,
Travis Schluessler, Prashant Chandra, Adrian Georgescu
Intel Communications Group, Intel Corporation

Index words: software framework, network processor, data plane, microblocks

ABSTRACT

Network processors are being targeted at a wide range of applications with varying packet processing and throughput requirements. The programmability and flexibility of a network processor make it suitable for applications ranging from Voice over IP to mobile IPv6 with data rates spanning OC-3 to OC-192. In such an environment, the investment made in software development by equipment manufacturers is increasingly significant. Preserving this investment and leveraging it across multiple projects are key considerations when making the right choice of a network processor.

The IXP family of processors provides a very powerful and scalable solution for the network processor market. The IXP2400 targets data rates from OC-3 to OC-48, while the IXP2800 is designed for applications with data rates ranging from OC-48 to OC-192. The IXA portability framework provides the associated software infrastructure to help develop modular, reusable software building blocks for these processors.

This paper describes how the IXA portability framework helps accelerate software development and improves code reuse across applications. The paper details how applications written to the IXA portability framework may be scaled up or down to fit the needs of a specific application.

The paper describes a reference application–IP forwarding with DiffServ–and its implementation in a number of different configurations including dual-chip IXP2400 at OC-48 rates, single-chip IXP2400 at 2 x OC-12 rates and single-chip IXP2800 at 2 x OC-48 rates.

The paper illustrates how the design of the application may be structured to facilitate reuse of software blocks

across these configurations. Using this design as an example, the paper describes various features of the IXA framework and its value in the software development cycle.

The paper assumes that the reader is familiar with the Intel Network Processor Family [1].

INTRODUCTION

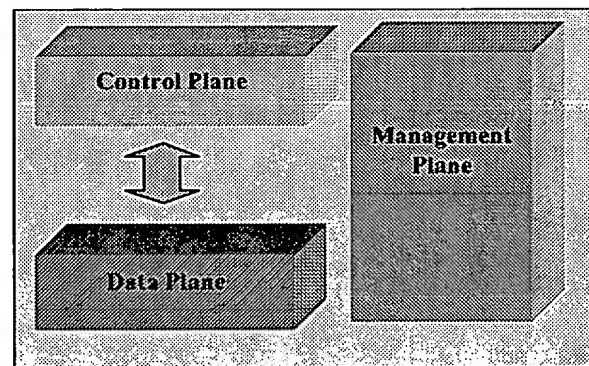A networking application typically operates on three logical planes (Figure 1):

Figure 1: Logical planes in a networking application

1. The *data plane* processes and forwards packets at high speed. It is typically the most performance sensitive since all packets processed by the device must pass through here. In the IXP family, the data plane consists of

- the *fast path*, i.e., the MEv2 microengines, which handle most of the packets. For example, the fast path handles the simple forwarding of Ipv4 packets.

- the *slow path*, i.e., the XScale™ core. This handles a few packets that cannot be handled on the fast path because of the complexity of the processing involved.

These packets are called *exception packets*. Examples include forwarding IP packets with options in the header, packets that need to be fragmented, etc.

2. The *control plane* handles protocol messages and is responsible for the setup, configuration, and update of tables and data sets used by the data plane for lookups. For example, the control plane processes Routing Information Protocol (RIP) packets and updates the IPv4 forwarding table used by the data plane. In the IXP environment, the XScale core may function as the control plane, or this functionality could be supported by an external processor.

3. The *management plane* is responsible for system configuration, gathering and reporting statistics, and stopping or starting applications in response to user input or messages from other applications.

The IXA portability framework is used primarily to develop data plane software and to help interface with code running on the control plane.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

FAST PATH: MICROENGINE FRAMEWORK

Developers writing microengine code must consider the tradeoffs between modular, easy-to-maintain, and reusable code versus performance. For bleeding edge applications, it may be important to get the highest level of performance possible. For applications in which new features will be added over time, modularity and code maintenance are likely to be more important.

The IXA portability framework is intended to help address both these needs for customers by offering a tiered solution where the programmer can use as much of the framework as appropriate for the application. As more of the framework is included, the developer will get more modular and reusable code.

The IXA portability framework consists of different layers that the developer can choose to incorporate into the design. One layer consists of the tools including support for a high-level programming language. The next layer is a set of libraries that are optimized for the particular network processor. The last layer is a programming model called microblocks.

We describe each of these layers in the following sections.

High-Level Language Support: Microengine C Compiler

At the lowest level of the framework are the tools. The IXA portability framework provides both an assembler and a C compiler for the microengines. By providing a high-level language through C, the framework helps to abstract some of the underlying hardware details from the programmer. There are significant advantages to using C over microengine assembly.

- C is the programming language of choice for most embedded system and network application developers. Availability of a C compiler reduces the learning curve for programmers new to the IXP environment.

- A high-level language is much more effective at abstracting and hiding the details of the microengine instruction set from programmers.

- It is typically easier and faster to write modular code and maintain it in a high-level language with its superior support for data types, type checking, etc.

However, since code running on the microengines is on the packet processing critical path, it is extremely important to generate very efficient code that meets the performance requirements of the application.

Intel's Microengine C Compiler is specially optimized for the IXP2400 and IXP2800 microengine environment. For example:

- The compiler allows the programmer to specify which variables must be stored in registers and which may be stored in memory.

- The compiler allows the programmer to specify the type of memory (off-chip SRAM, DRAM, or on-chip scratch memory) that should be used to allocate a specific variable or data structure. Each type of memory has different latency and access characteristics, and the compiler gives developers more control in laying out the data structures across different types of memory.

- To handle hardware specific features, the compiler supports intrinsics and inline assembly. The use of these, however, undermines the portability of the code to future generations of network processors. In this case, the data plane libraries provide the only isolation from the hardware and are critical for code portability. For this reason, the IXA portability framework supports the data plane libraries in both microengine assembly and microengine C.

- The compiler has a packet format for bitfield structures. Unlike standard C bitfield structures, where the fields must line up with a 32-bit boundary, the fields of a packed structure in microengine C have no such restriction. This is highly suitable for defining and accessing fields of protocol headers.

In summary, the Microengine C Compiler allows developers to write code in a high-level language while still meeting the performance expectations of fast-path packet processing.

Low-Level Support APIs: Data Plane Libraries

The next component of the IXA portability framework consists of the data plane libraries. These are libraries that abstract out some of the implementation details of the hardware, as well as provide common building blocks that many applications will need. These libraries are provided as assembler macros and C functions.

The data plane libraries are optimized for maximum performance and minimal code space utilization. They are the foundation for all MEv2 microengine code. The libraries consist of several portions that enhance code portability and reuse (Figure 2).
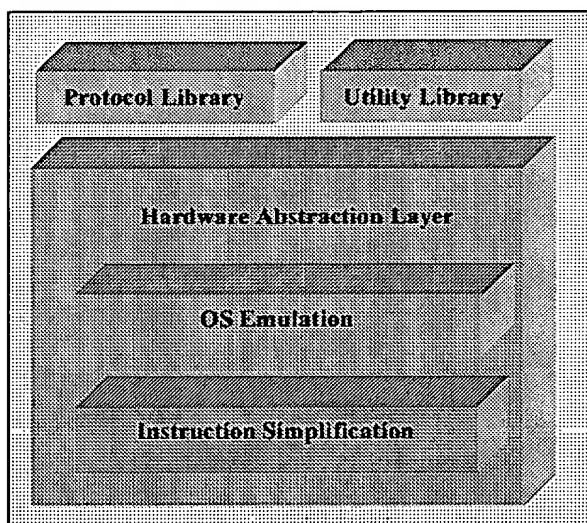

Figure 2: Low-level libraries on the microengine

The instruction simplification library provides assembler macros that simplify common microengine operations such as loading constants, accessing Chip-Specific Control and Status Registers (CSRs), and performing indirect operations. This library isolates developer and the software they write from changes to the instruction set. The use of the data plane library ensures that the code is not only portable to the new processor but also able to perform better with the latest instruction set.

The Operating System (OS) emulation library contains support for services that would normally be provided by an OS. This library uses the underlying hardware to support inter-thread signaling, messaging, synchronization (locking and critical sections), queuing, buffer management, timers, etc. As with instruction simplification, the use of the library isolates the programmer from changes to future hardware implementations.

The utility library includes support for hash table lookups, endian swaps, CRC (Cyclic Redundancy Check), and other useful functions.

The protocol library provides an optimized implementation of many common networking functions. Some examples of these are functions that parse and modify IP packet headers.

Modular Building Blocks: Microblocks

The highest layer of the IXA portability framework for the microengines is the programming model and associated support. In this programming model, the developer divides the fast-path processing of the application into high-level logical components called *microblocks*.

The programming model enables and requires microblocks to be written such that each microblock is independent of others. By providing clean boundaries between these blocks, the programming model makes it possible to modify, add, or remove more microblocks without affecting the behavior of the other blocks. This improves reusability and allows developers to combine microblocks and create different applications. The net benefit is that the task of writing fast-path code is simplified and time-to-market is accelerated.

Each microblock is an assembler macro or C function written using the underlying low-level data plane libraries. Note that as opposed to a low-level function, e.g., ipv4_checksum(), a microblock is coarse-grained and has state and associated data structures typically shared with the XScale™ core. Examples of microblocks include IPv4 forwarding, Ethernet layer-2 filtering, 5-tuple classification, Multiprotocol Label Switching (MPLS) label insertion, etc.

A microblock has an associated management component on the XScale core. The application is typically written so that the microblock will process most common cases and pass exception cases to the XScale component for further processing.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

DESIGNING FOR MODULARITY AND REUSE

Figure 3 shows an IP DiffServ application implemented using microblocks. We will use this application to illustrate some important design considerations for modular code.
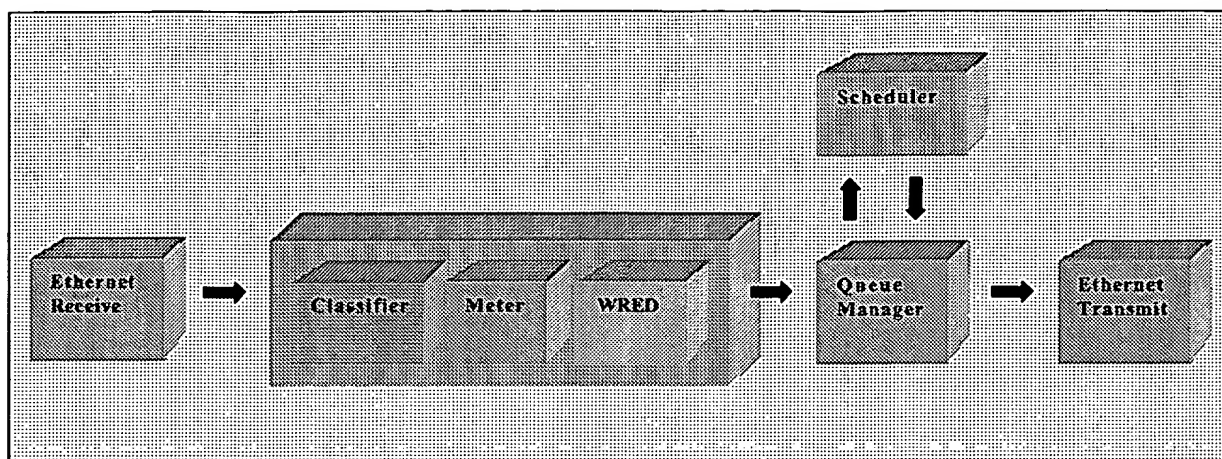
Figure 3: Single-chip DiffServ application using microblocks

Separate Application-Specific Code from Reusable Code

Applications written to the IXA framework include modular reusable microblocks and glue code that combines these blocks and implements the data flow between them. This glue code is referred to as the *dispatch loop*. The dispatch loop combines microblocks running on the same microengine thread and implements the packet data flow between them.

The same dispatch loop may be instantiated on more than one microengine thread to process multiple packets at the same time. For example, in Figure 3, the classifier, meter and Weighted Random Early Detection (WRED) blocks are combined by a dispatch loop and run on several microengine threads in parallel.

In the IXA framework, application-specific code is implemented in the dispatch loop, while care is taken to preserve the reusability of microblocks across applications.

Support for Shared State between Microblocks

An important aspect of the framework is the mechanism by which state is shared between microblocks. This includes per-packet state, referred to in IXA terminology as *packet meta-data* (packet size), offset to the packet data in memory, destination port, etc. Another example of shared state is packet headers that are read and modified by different microblocks in the pipeline. In the IXA framework, the dispatch loop caches packet meta-data and other shared state in registers and provides access to them through an API. The dispatch loop also supports caching packet headers in local memory or registers. Across microengines, the cached information is flushed to memory or passed along in the message queue between

the microengines. The dispatch loop populates the cache by reading information either from memory or from the message queue. This isolates the microblocks from the layout of the meta-data in memory and from the format of the messages on the message queue between microengines. In addition, the packet header cache and meta-data cache considerably improve performance.

For example, in the DiffServ application in Figure 3, the dispatch loop reads in the IP header from memory and caches it in local memory. This header is then used by the classifier to do a lookup and determine the flow and class information for the packet. The same header is also used by the meter stage, which modifies the TOS/DSCP field in the IP header and recalculates the checksum. The modified packet header is written back by the dispatch loop after the WRED microblock. The dispatch loop caches the packet header and per packet information such as flow and class id in registers. The microblocks (classifier, meter, WRED) simply access these through well-defined API's exported by the dispatch loop.

Decouple Application Data Flow from Microblock Design

Typically, the classification and processing performed in a microblock determines to which microblock the packet goes next. For example, based on the processing in a filter block, the packet may either be sent to an IPv4 forwarding microblock or be dropped.

Care must be taken to decouple this data flow from the design of a specific microblock. To preserve its reusability across applications, a microblock must not require a specific microblock to be run before or after this block. In the IXA framework, the dispatch loop implements the data flow between microblocks. Each microblock indicates only the logical result of its classification.

For example, the filter block indicates the next block-PASS or DENY using a dispatch loop state variable. Based on this variable, the dispatch loop may decide that if the next block is PASS, the packet is sent to the IPv4 forwarding microblock, and if it is DENY, the packet is dropped. Alternatively, the dispatch loop may be modified such that instead of dropping packets that are denied, these packets are sent to the XScale™ core.

Separating Packet-Processing Blocks from Network Interface-Specific Blocks

In any networking application, we can clearly differentiate between blocks that are hardware or network interface specific and blocks that process packets.

Network interface-specific blocks include the receive and transmit blocks for the different media interfaces (POS, ATM, Ethernet, etc). Other hardware-specific blocks include the queue manager block that handles the queuing hardware on the IXP2400/IXP2800. Packet-processing blocks are protocol specific, e.g., IPv4/v6 forwarding, Network Address Translation (NAT), layer-2 bridging, etc.

Network interface-specific blocks (receive and transmit) are typically involved in segmentation and reassembly of a

packet. It is a good design practice to decouple the packet-processing microblocks from the segmentation/reassembly of packets by placing them on separate microengines.

For example, in the IP DiffServ application shown in Figure 3, the classifier, meter and WRED blocks are placed on separate microengines from the receive block and interface to it using a message queue.

This has a couple of advantages:

- It builds some elasticity into the packet-processing pipeline and allows sudden bursts of received data to be absorbed. For example, typically, the packet pipeline is expected to sustain an average rate lower than the peak arrival rate of traffic. With this design, only the receive/transmit blocks need to be able to sustain the peak rate. As long as the elasticity provided by the message queue between microengines is sufficient, the packet-processing blocks can run at a lower rate.

- The packet-processing blocks may be modified or replaced easily without affecting the rest of the pipeline. Figure 4 shows how the packet-processing blocks in the DiffServ application shown in Figure 3 may be replaced by an IPv4 forwarding block, while reusing the rest of the blocks in the pipeline.

- The network interface blocks may change in future revisions of the processor. For example, future revisions of the processor may implement the reassembly in hardware. This design hides these hardware changes from the packet-processing code.
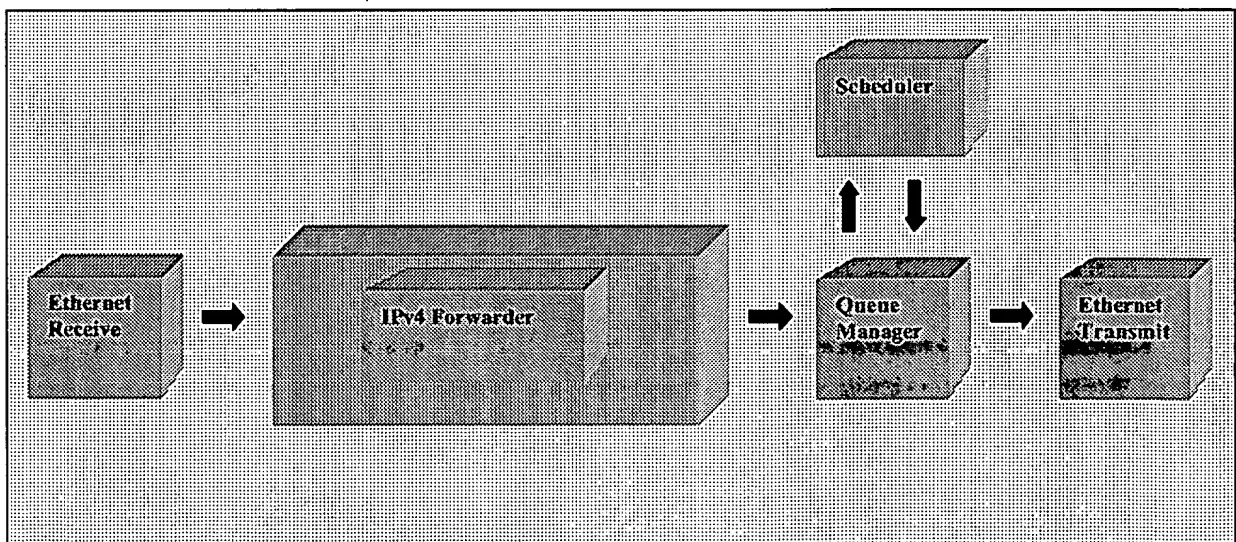

Figure 4: IPv4 forwarding application using microblocks

- The network interface blocks may be different depending on the media and bus configurations. These blocks are also slightly different between IXP2400 and IXP2800. The packet-processing blocks, on the other hand, remain the same in all configurations. By separating the two, the design allows for maximum reuse of code.

- Since the receive/transmit blocks may need to sustain bursts of traffic, they need to be highly optimized. For this reason, they may be implemented entirely in assembly. By moving the packet processing code to a different microengine, we eliminate the need to mix assembly and C code while combining blocks.

While this approach has the many advantages indicated above, it also implies an extra read and write of the packet header when a minimum size packet is received. For this reason, applications that target full-line rate at OC-192 data rates for minimum size packets may choose to combine the packet-processing code with the packet reassembly [2].

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

SLOW PATH: XSCALE™ CORE FRAMEWORK

In the IXP environment, the XScale may be used for management, control, and even packet processing (Figure 5).

A *core component* is the slow path or XScale counterpart of the microblock and is responsible for its configuration and management. In addition, core components may handle packets that cannot be processed on the microengines because of the complexity involved.

The IXA portability framework provides the support infrastructure and APIs required to develop core components.

Network application developers are commonly faced with the challenge and opportunity to leverage existing code for the slow path and interface it with code written to the IXA portability framework. For this reason, the services provided by the framework on the XScale are divided into discrete layers of functionality. At the lowest layer, the IXA core framework supports a set of APIs called the *resource manager APIs*. These APIs provide support for hardware initialization, configuration, and resource management. They also support communication between the microengines and code running on the XScale. The resource manager APIs isolate the XScale developer from

the specifics of hardware and from the details of microengine to XScale communication.

While a slow path application may be written entirely to the resource manager APIs, the IXA framework also defines a standard modular way of writing XScale core components. The *core component infrastructure library* defines the structure and canonical design of an XScale core component and the mechanism by which messages and packets are passed between core components. Architecturally, a clean interface separates a core component from the rest of the system.
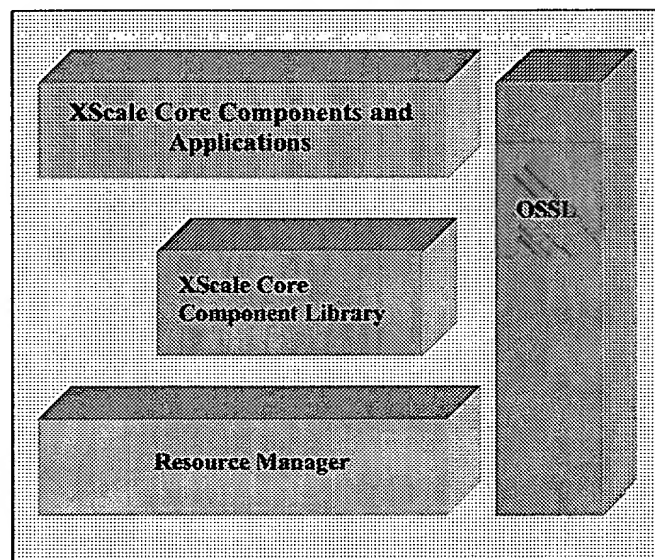

Figure 5: XScale core framework

In most cases, a core component manages a single microblock. The core component/microblock pair may be viewed by the control plane and the rest of the data plane as a single unit of packet processing and reused in a variety of processing configurations. Viewing the pair as a single unit allows an intelligent and highly flexible split of packet processing between the core component and the microblock, depending on the needs of specific applications.

Another important aspect of code reusability on the XScale core is abstraction from the operating system (OS) involved. Depending on the application, the same network vendor may choose a small footprint proprietary micro-kernel for one product and a much more mature industry standard Real-Time OS (RTOS) for another.

For this reason, the IXA Framework defines an OS Services Layer (OSSL) that provides APIs for commonly

used OS services (e.g., timers, threads, semaphores, etc). Consistent use of this API greatly enhances the portability of the framework (and code written to the framework) to different operating systems.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

INTERFACING WITH THE CONTROL PLANE

Data plane processing is controlled and managed by control plane software and stacks. In most legacy systems, these typically reside on a separate processor and exchange protocol packets and control messages with the data plane over a control bus or a backplane.

On the IXP family of network processors, some of that functionality may be moved to the XScale™ core. It is important, however, that the control plane is logically separated from the data plane. If the two planes are separated by a set of standard APIs and protocols, equipment manufacturers can choose and upgrade control stacks independent of data plane software. These stacks may come from any vendor who supports these APIs. Upgrades of network processors and migration to newer generations will not require any changes to control plane software.

The Network Processor Forum (NPF), of which Intel is a prominent member, is actively working on defining such standards and APIs. The NPF API that is emerging from this body abstracts the data plane as seen by the control plane and defines per-protocol management interfaces between the two planes.

A working group within the Internet Engineering Task Force (IETF) called Forwarding Control Element Separation (ForCES) is in the process of defining a messaging protocol for control plane or data plane communication. The interconnect-independent nature of this protocol makes it easy to change the bus or the backplane without affecting the software layers that are above this protocol. In fact, the control plane and the data plane software may become located on the same network processor with no changes required, other than to the implementation of the ForCES protocol.

The IXA portability framework provides a Control Plane Product Development Kit (CP-PDK), which supports industry standards such as the ones described above and enables data plane software to interface with the Control Plane.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

SCALING APPLICATIONS TO HIGHER DATA RATES

This section describes how the IXP family of network processors allows an application to be scaled to higher data rates and the challenges involved.

Moving from Single-Chip to Dual-Chip Configurations

A common way to scale an application to higher data rates is to use more microengines running in parallel. The IXP2400 and IXP2800 may be used in both single-chip and two-chip configurations, thereby allowing the developer to increase the number of microengines available.
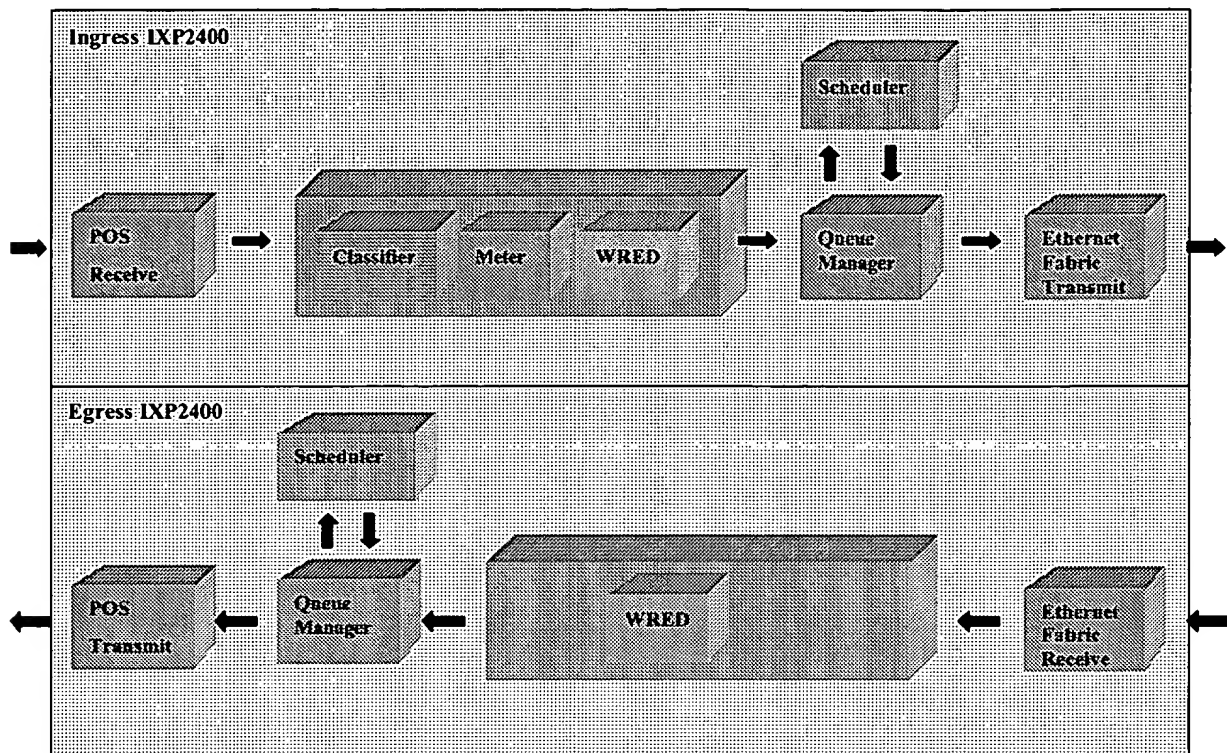
Figure 6: OC-48 DiffServ with dual-chip IXP2400

Figure 6 shows the implementation of DiffServ at OC-48 data rates using two IXP2400 processors. The ingress processor receives from a POS network interface and transmits into an Ethernet fabric. The egress processor receives from the Ethernet fabric and transmits into the POS network interface.

The same application may be run on a single IXP2400 chip at OC-24 data rates, simply by combining the pipelines on the ingress and egress processors onto a single processor. For example, in the single-chip pipeline a single microengine would implement both the POS receive and transmit processing using four threads for each function.

Challenges

There are some limitations to this approach that must be addressed and overcome at design time.

- Combining blocks onto a single microengine may not always be possible due to conflicts in usage of resources like CAM and local memory. For example, it may not be possible to combine two schedulers onto a single microengine because both make extensive use of local memory. Similarly, combining the ingress and egress queue managers may not be possible since both use the CAM. However, the developer can combine the scheduler and the queue manager blocks

into a single microengine, since the scheduler does not use the CAM and the queue manager does not use too much local memory.

- Another possible problem while combining blocks onto a single microengine is to run out of instruction code store. Again, the only solution is to intelligently pick and combine blocks for which this is not an issue.

Moving from IXP2400 to IXP2800

While the IXP2400 can handle up to OC-48 data rates, the IXP2800 provides the upgrade path to 2 x OC-48 with a single-chip configuration and OC-192 with a dual-chip configuration.

Moving an application from IXP2400 to IXP2800 is greatly simplified by the fact that the instruction set and microengine architecture are identical. The IXP2800 runs at a higher clock frequency (1.4GHz versus 600MHz for the IXP2400) and supports a greater number of microengines (16 versus 8 on the IXP2400).

. Table 1: Comparison of the IXP2400 and IXP2800 for a DiffServ application

Parameter	IXP2400 single chip	IXP2400 dual chip	IXP2800 single chip	IXP2800 dual chip
Line rate	1.2 Gbps	2.408 Gbps	2 x 2.408 Gbps	4 x 2.408 Gbps
Min POS packet size	49	49	49	49
Packet throughput	3.071 million packets/sec	6.14 million packets/sec	12.28 million packets/sec	24.57 million packets/sec
Clock frequency	600 MHz	600 MHz	1.4 GHz	1.4 GHz
Inter-packet arrival time	195.37 cycles	97.68 cycles	114 cycles	57 cycles
Compute cycles per packet for a context pipe stage (microengine)	195.37 cycles	97.68 cycles	114 cycles	57 cycles
Latency per packet for a context pipe stage (microengine)	195.37*8 cycles	97.68*8 cycles	114*8 cycles	57*8 cycles
Compute cycles per packet for a functional pipeline of n microengines running in parallel	195.37*n cycles	97.68*n cycles	114*n cycles	57*n cycles
Latency per packet for a functional pipeline of n microengines running in parallel	195.37*n*8 cycles	97.68*n*8 cycles	114*n*8 cycles	57*n*8 cycles

Table 1 compares the per-packet instruction count and latency constraints for an IXP2400 targeting single-chip 2xOC-12 and dual-chip OC-48 versus an IXP2800 targeting single-chip 2xOC-48 and dual-chip OC-192.

The table indicates that an IXP2800 running in single-chip mode has approximately the same number of instruction cycles (114 cycles) to process a minimum-size POS packet as an IXP2400 in a dual-chip configuration (97 cycles).

This, along with the fact that a single-chip IXP2800 has the same number of microengines as a two-chip IXP2400 (Table 2), implies that at least at an instruction count level, an application running at OC-48 rates on a dual-chip IXP2400 will scale to 2xOC-48 rates on a single-chip IXP2800. Similar considerations may apply to the OC-192 application running on a dual-chip IXP2800 with 32 available microengines.

CHALLENGES

While the code reuse between applications running at the different data rates is good, there are some limitations that must be considered at design time when moving from an IXP2400 to the IXP2800.

- Some blocks cannot be run in parallel on more than one microengine. So the availability of more microengines does not help. The queue manager is an example of such a block. It manages the queuing hardware on the IXP2800 using the CAM local to the microengine. Since it cannot be executed in parallel on more than one microengine, the code for the dual-chip IXP2800 must be optimized to fit the OC-192 (57 cycle) instruction budget.

- Even though the IXP2800 runs at a higher clock frequency, the memory speed does not scale up the same way. This implies that memory table accesses (relative to the number of instructions executed) take longer on the IXP2800.

- There are some differences in the Media Switch Fabric (MSF) between IXP2400 and IXP2800. The differences are minor and may be handled with processor-specific switches in the receive and transmit code.

Table 2: Microengine allocation for the DiffServ application

Function	IXP2400 single chip	IXP2400 dual chip	IXP2800 single chip	IXP2800 dual chip
POS receive	½	1	1	2
POS transmit	½	1	1	2
Classifier/meter/WRED	2	4	6	8
Ingress queue manager	½	1	1	1
Egress queue manager	½	1	1	1
Ingress scheduler	½	1	1	2
Egress scheduler	½	1	1	2
Egress WRED block	½	1	2	4
Ethernet fabric receive	½	1	1	2

- The receive and transmit blocks run on a single microengine for the OC-48 and 2 x OC-48 cases, whereas they are executed on two microengines in the OC-192 case. This implies that some of the receive or transmit context stored in local memory may need to be flushed out to SRAM in the OC-192 case.

The implication of these challenges is that while it is very easy to get code written for an IXP2400 to work on an IXP2800, special optimizations may be needed to get the maximum performance from the chip.

CONCLUSION

When network equipment vendors select a network processor, they make a significant commitment to use it for years to come. It is important to ensure that the network processor environment selected is flexible and can be scaled to protect the vendor's investment. Software reusability and the tools and framework that enable it should be a key consideration when selecting a network processor.

The IXP family of network processors provides a powerful and scalable solution to the problem of programmable network devices. The IXA portability framework provides the associated software infrastructure to help develop modular and reusable software building blocks for these processors.

By providing the necessary infrastructure to help accelerate software development and by improving code reuse across applications, the IXA framework adds considerable value to Intel's network processor solution.

REFERENCES

[1] Matthew Adiletta, et al., "The Next Generation Family of Intel Network Processors," *Intel Technology Journal*, Vol. 6 issue 3, August 2002.

[2] Matthew Adiletta, et al., "Packet over SONET: Achieving 10 Gigabit/sec Packet Processing with an IXP2800," *Intel Technology Journal*, Vol. 6 issue 3, August 2002.

[3] S. Blake, et al, "An Architecture for Differentiated Services," IETF RFC 2475, December 1998.

AUTHORS' BIOGRAPHIES

Uday Naik is a senior staff software engineer at Intel's Network Processor Division. His professional interests include networking, embedded systems and digital television. Uday received his Master's degree in Computer Science from the University of Indiana Bloomington in 1992. He also holds a Bachelor's degree in Computer Science and Engineering from the Indian Institute of Technology (IIT) Bombay. Uday resides in Fremont, California, and can be reached via e-mail at uday.naik@intel.com

Larry Huston is a principal software architect at Intel's Network Processor Division. He is responsible for defining the software requirements for future network processors,

as well as designing the advanced programming framework. Prior to Intel, Larry was a software architect at NetBoost, where he helped design their programming environment for accelerating network applications such as firewalls and intrusion detection. Prior to NetBoost, Larry was a member of the technical staff at Ipsilon Networks, where he designed and implemented Ipsilon's protocols for distributed IP switching and forwarding. Larry received his Ph.D. degree in Computer Engineering from the University of Michigan in 1995. He also holds M.S.E. and B.S.E. degrees in Computer Engineering and Aerospace Engineering from the University of Michigan. Larry can be reached via e-mail at larry.huston@intel.com.

Prashant Chandra is a senior staff network architect in the Network Processor Division at Intel Corporation. His interests are in the areas of programmable networks, signaling protocols, and traffic management. He received his B.E degree in Electronics Engineering from Bangalore University in 1991, an M.S. degree in Computer Engineering from West Virginia University in 1994, and a Ph.D. degree in Computer Engineering from Carnegie Mellon University in 2000. His e-mail is prashant.chandra@intel.com

Donald Hooper is a senior software architect in the Network Processor Group. He has led many projects including logic synthesis, video servers, MPEG-2 and DAVIC standards, IXP1200 software tools and libraries, IXP2800 proof of concept designs, and NPG coding standards. His professional interests include networking, artificial intelligence, and object-oriented languages. He attended four colleges with cumulative B.S.E.E. credits finishing at UCLA. He resides in Shrewsbury Massachusetts. His e-mail is donald.hooper@intel.com

Travis Schluessler is an engineering project lead at Intel's Network Processor Division. Professional interests include networking and embedded systems. Travis received his Bachelor's degree in Electrical and Computer Engineering from Carnegie Mellon University in 1992. He resides in Cupertino, California, and can be reached via e-mail at travis.schluessler@intel.com.

Adrian Georgescu is a staff network software engineer at Intel's Network Processor Division. Adrian received a Master's degree from Polytechnic Institute, Bucharest, Romania, in Aerospace Engineering. His professional interests are networking, OO programming, and numerical algorithms. Currently, he lives in Palo Alto, California, and can be reached by e-mail at adrian.georgescu@intel.com

Duke Tallam is a software engineering manager in the Network Processor Division at Intel Corporation. Duke has over 20 years of software development experience in a wide variety of fields spanning deeply embedded to huge turnkey systems. Duke received his Master's degree in Electrical Engineering from South Dakota School of Mines and Technology in 1983. He holds a Bachelor's degree in Electronics from Bangalore University. Duke resides in Fremont, California, and can be reached via e-mail at duke.tallam@intel.com.

Alex Shoykhet is a senior software product manager at Intel's Network Processor Division. He can be reached at alex.shoykhet@intel.com

Raj Yavatkar is chief software architect for Intel's Network Processor Group. He can be reached at raj.yavatkar@intel.com

Network Processor Building Blocks for All-IP Wireless Networks

Harsh Vipat, Philip Mathew, Manohar Ruben Castelino, Auro Tripathy
Intel Commuications Group, Intel Corporation

ABSTRACT

The focus of this paper will be on the hardware features of the IXP2400 network processors and how they help to accelerate the key processing needs of wireless networks. The first part of the paper will explain the 3G wireless network topology and the role of network nodes such as base stations, radio network controllers, and routing gateways. The next part of the paper will identify a wide range of packet processing functions performed at the radio network controller (RNC) node. The heart of the article will delve into the inter-workings of seminal reusable network-processor-based building blocks such as packet forwarding at layer-3, bandwidth-saving header compression and decompression schemes, IPv6-to-IPv4 tunneling, and QoS. The article will conclude by asserting that all this, coupled with programmability and hardware acceleration capability, meets the evolutionary needs of wireless networks.

INTRODUCTION

Traditional wireless telecommunication networks and data communication networks will converge. The most recent specification of the converged network (also known as 3GPP Release 5) specifies a wireless network where the transport layer utilizes Internet Protocol (IP) networking as much as possible. In this all-IP network, both user data flows and control flows will be based on IP, thus making the end-to-end network a packet-switched IP network. In practice, the single most important packet data protocol to be supported is IPv6. A simplified reference model for the General Packet Radio Service (GPRS) network is shown in Figure 1.

The role of the mobile terminal in an all-IP network is to originate and terminate both connection-oriented (TCP/IP) and connectionless (UDP/IP) real-time and non-real-time services such as web browsing and VoIP calls.

The rest of the components of a wireless network can be broadly categorized into the Radio Access Network (RAN), comprised of the Base Station and the Radio Network Controller (RNC) and the Packet-Switched Core Network comprised of the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN).

Base stations (also known as Node Bs) link the mobile terminal to the rest of the fixed and mobile network. Each base station provides radio coverage to a geographical
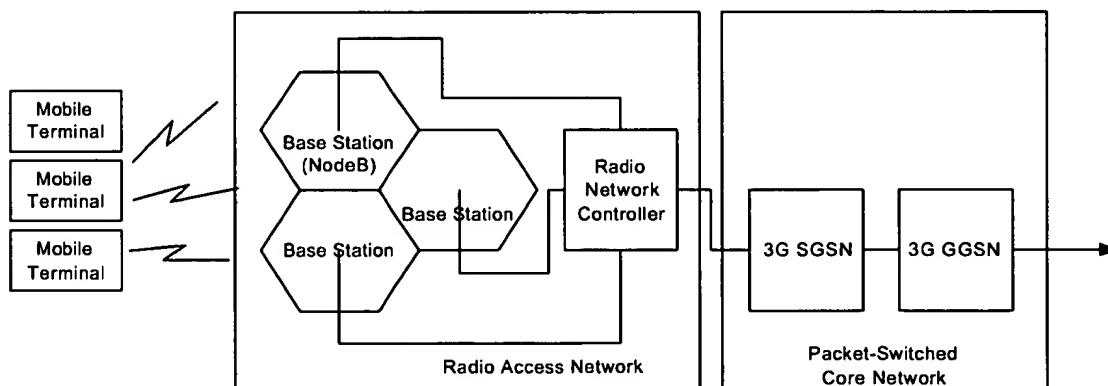

Figure 1: Nodes of 3G wireless network

area known as a cell. A typical network may be visualized as consisting of a mesh of hexagonal cells, each with a base station at its center. Base stations are connected to one another by central switching centers known as Radio Network Controllers (RNCs). The RNC (Figure 2) manages the wireless radio interfaces of the base stations and controls handoff, sending data from the core network to one or more base stations in the forward direction, and selects the best signal from several base stations and sends it to the core network in the reverse direction. For example, if a mobile terminal user moves out of one cell and into another, the RNC hands over communication to the adjacent base station (the switching function). Alternately, the RNC may route calls (packets) to another RNC in the network.

In the core network, the SGSN takes care of routing, handover, and IP address assignment. For example, if you were in a car on the highway and were browsing the Internet on a mobile terminal, you would pass through many different cells. The SGSN routes the packets to the appropriate base station and maintains a seamless connection.

The GGSN is the "port of last call" in the Core Network before a connection to an ISP or corporate network's

router occurs. The GGSN is basically a gateway, router, and firewall rolled into one.

While the packet processing needs of various nodes in a wireless network are unique, the processing needs of the Radio Network Controller (RNC) are most challenging. The RNC serves as a transition point between the predominantly IPv6 Radio Access Network (RAN) and the Core Network (CN), which has both IPv6 as well as IPv4 traffic. The RNC also handles both packetized voice and data packet flows, which have different requirements in terms of delay and loss characteristics. The processing functions at the RNC therefore include IPv6 routing, IPv4 routing, header compression and decompression, tunneling and QoS. All these functions can be implemented as reusable building blocks on IXP2400.

The rest of the paper is organized as follows. First, we take a brief detour and provide overview of the IXP2400 network processor and a possible software architecture for implementing the building blocks. The remaining sections cover the processing functions of RNC nodes and highlight the specific hardware features of IXP2400 that can be utilized to implement them efficiently.
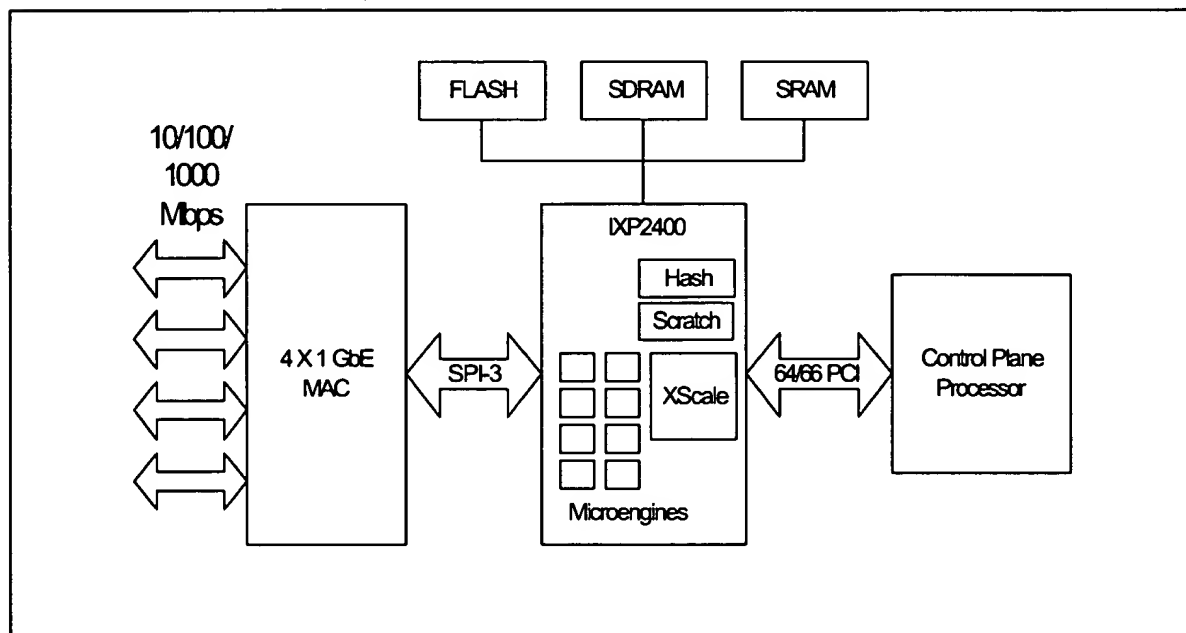

Figure 2: Conceptual block diagram of IXP2400-based RNC

OVERVIEW OF THE IXP2400 PROCESSOR

The IXP2400 has eight programmable second-generation microengines (MEv2) and an integrated XScale™ core. The common processing, called fast-path processing, performed on a majority of the packets, is implemented on microengines. Housekeeping functions and exception processing (performed on a minority of the packets) are typically implemented on the XScale.

Each microengine has eight threads, and each thread has its own hardware context consisting of a register set, program counter, etc. The context swaps are therefore very inexpensive. The threads are scheduled non-preemptively by the hardware. The microengines also have a fully associative 16-entry CAM and 640 32-bit words of local memory to speed up stateful packet processing.

On a network processor, the packets can arrive at a specified maximum line rate. The processor must perform the required processing on these packets and must transmit the processed packets in sequence at the desired rate. When the time required to process each packet far exceeds the inter-packet arrival time, a software pipeline architecture can be deployed to achieve required line rates.

In a software pipeline model, the processing required for a packet is divided into several sequential stages. Each stage provides only a part of the entire processing, and the packets therefore go through all the stages to complete entire processing. There are two basic pipelining approaches: context pipelining and functional pipelining. In a context pipeline, each stage is implemented on a microengine, and each microengine therefore works on different stages of a packet. In functional pipelining, the same microengine processes different stages of a given packet.

Fundamental to implementing software context pipelines across microengines are IXP2400 features such as hardware-assisted scratch-memory resident producer/consumer rings (referred to as scratch rings) and private registers between adjacent microengines (referred to as next-neighbor register rings) that allow efficient implementation of producer and consumer communication. These hardware-managed mechanisms rapidly pass state from one microengine to the next. The IXA SDK provides a framework for implementing processing functions as reusable building blocks (microblocks) and for combining them in desired fashion to form functional pipelines. The following sections describes how the functions of RNC nodes can be implemented using a combination of context and functional pipelining.

PROCESSING REQUIREMENTS OF RNC NODES

The packet processing in an RNC node (Figure 3) can be divided into four major stages: the receive stage (Rx), the header processing stage, the QoS stage, and the transmit stage (Tx). The receive stage is responsible for layer-2 reassembly and framing. The header processing stage can include a variety of functions such as layer-3 forwarding, header compression, etc. The QoS stage provides priority queuing of packets based on classes. The transmit stage is responsible for transmission of the frames. Each of these steps can be implemented as a context pipeline stage on a set of microengines on IXP2400.

The Rx and Tx functions are link-layer specific and can be easily implemented with the help of features provided by the media and switch fabric interface available on IXP2400. The header processing and QoS stages involve more complex and stateful processing and will be the focus of following subsections.
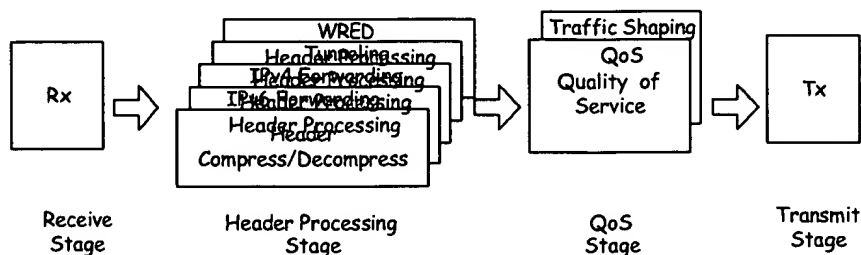
Figure 3: RNC processing stages

Header Processing Stage

Figure 4 shows the block diagram of the header processing stage. The header processing stage can be implemented as a functional pipeline, and each processing function is therefore implemented as a microblock. The following sections describe details of each block and the hardware features that are relevant for the efficient implementation of the block.

Link-Layer Decapsulation and Classification

Since the RNC connects the radio networks to the core network, the packets arriving at the RNC may have different link-layer encapsulations. This poses a challenge for the next stage, namely, header processing. All the processing blocks in the header-processing stage primarily process layer-3 and higher layer headers. Due to different sizes of link headers, the offset of the layer-3

are the byte_align instruction, index mode addressing of registers, and local memory. The byte_align instruction allows concatenation of data in two 32-bit registers and extraction of any four bytes from a concatenated string into the destination register. The byte index for alignment can be selected by setting a control status register (CSR). The IXP2400 also allows indirect referencing of the transfer registers. An index pointer can be set to point to a particular transfer register by writing to a control status register. The transfer register can then be accessed by indirect referencing through the index pointer. Auto increments and decrements of the index pointer are also supported.

The local memory is addressable storage located in the microengine. The local memory can be accessed at long-word (32-bit) granularity, and the latency cost of an access is the same as accessing general-purpose registers. Two

Figure 4: Flow diagram of the header processing stage
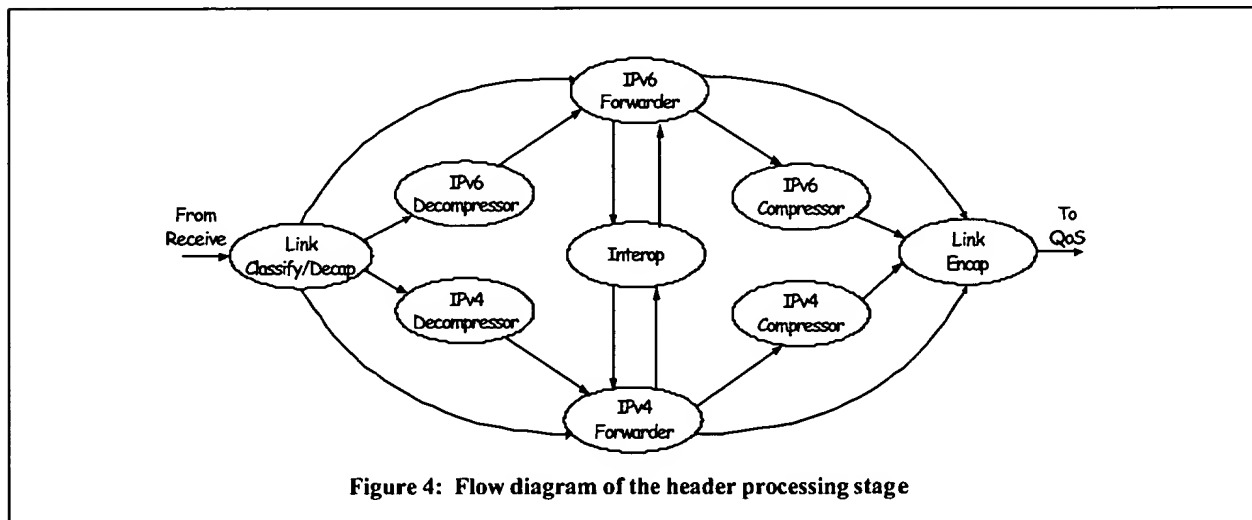
header can only be determined at run-time. This unnecessarily requires each processing block to know about different link encapsulations and deal with unaligned headers. The link classify block solves this problem by aligning and caching the layer-3 and higher headers in the local memory of the microengine. This is advantageous in many ways. The processing blocks can access the header fields with a minimum of one cycle latency. The headers can be treated as global data structures and can be shared by all the processing blocks in the pipeline without the copying and aligning overhead that each block downstream must undertake. Lastly, this approach also makes the processing blocks link-layer independent and enhances their reuse potential.

The key architectural features that allow efficient implementation of the aligning and caching functionality

index registers are available to the programmer, which can be set to point to any of the 640 long-words of local memory. The local memory can be accessed by de-referencing the index register, or by specifying offset using array notation.

IPv4 and IPv6 Layer-3 Forwarding

The RNC has to route packets between the predominantly IPv6 RAN and the Core Network (CN), which has both IPv6 as well as IPv4 nodes. The IPv6 and IPv4 forwarding are therefore the key functions of the header-processing stage. The aggregation of IP address space requires use of a special search technique called Longest Prefix Match (LPM). Routing prefixes (route entries) are stored in a route table along with their associated next-hop forwarding information. The route table is searched to find

a longest prefix that matches with the destination address. The next hop information associated with the matched routing prefix is used to forward the packet. In order to facilitate LPM, routing prefixes are usually stored in complex tree-like data structures in SRAM. As a result, SRAM is accessed several times during the LPM.

Hardware-assisted multi-threading available on IXP2400 processors allows efficient implementation of memory-intensive algorithms such as LPM. A thread-issuing memory operation can explicitly yield control to other thread and allow some other processing to take place while the memory operation completes. The effective latency cost of the memory operations can thus be reduced or even be eliminated with the help of hardware-assisted multi-threading. Large numbers of GPRs allow efficient handling of long 128-bit IPv6 addresses. Since IPv6 supports hierarchical addressing and address aggregation in a structured way, the local memory can play an important role in optimizing the IPv6 LPM. Depending on the position of the router in the address hierarchy, the aggregation information can be cached in the local memory, and the route lookup can be speeded up substantially.

The layer-3 forwarders also have to handle a variety of exception conditions. These include processing of IP options, dealing with fragmentation and reassembly, processing dynamic route update requests, and responding to Address Resolution Protocol (ARP) or neighbor discovery messages. All these functions can be implemented on the XScale™ core. Once again, hardware-assisted scratch rings help in implementing communication between microengines doing the fast-path processing and the XScale core doing the exception path processing.

Header Compression and Decompression

Two factors contribute to the use of header compression schemes in wireless network. The first factor is that, for the IPv6 packet, the IPv6/UDP/RTP header is 60 bytes in size and a typical speech payload is about 20 bytes in size, a 300% overhead! The second factor is that wireless spectrum is expensive. Header compression schemes reduce the header to three bytes, yielding a manageable overhead of 15%.

Header compression schemes are based on the observation that many fields of the protocol headers rarely change during the life of a session. Also, many other fields change only in small, predictable quantities.

Compression and decompression are enabled by creating and storing a compression context for the RTP session at the compressor and the decompressor. A compression context has two parts: the context identifier and the

context information. The context identifier, a unique number denoting an RTP session, is derived from several fields of the header. For example, for RTP-based voice packets, the context identifier is derived from the source IP address, destination IP address, source port, destination port, and Synchronization Source Identifier (SSRC) fields. The context information includes all the fields in the IP, UDP, and RTP header. The very first packet of a session transmits the context identifier embedded in the uncompressed packet. Once the compressor and the decompressor get the context information associated with that context, compressed packets carry only the context identifier (pertaining to that RTP session) and the differences of the changing fields. The decompressor uses the context identifier to regenerate the full header.

In the IXP2400, multiple threads perform packet compression or decompression in parallel. The challenge is to ensure that the packets are exiting the compressor or the decompressor in the same order in which they entered. For every packet, threads need atomic access to the context information tables. This involves multiple accesses to high-latency external memory such as SRAM or SDRAM. Yet another factor slowing down performance is ensuring that packets are received by the threads in order, i.e., the first thread receives the packets, completes the critical section, and then signals the second thread, and so on. A method called *folding* described in the next section addresses this.

Folding

The principle of folding (or memory coalescing) provides that if a thread has already requested access to an external memory location (to be fetched into local memory), then other threads requesting access to that same memory location can simply wait to access it in local memory. In the meantime, they can yield to other threads. A combination of the microengine Content Addressable Memory unit (CAM unit) and local memory unit can be used to implement folding. A lookup into the 16-entry CAM results in a CAM miss or a CAM hit coupled with an index into local memory where the data resides. Thus, the microengine local memory acts as a cache, and the CAM aids in the cache lookup for every packet before accessing the external memory unit.

Folding can reduce accesses to external memory. The context identifier (uniquely identifying the context information of an RTP session) resides in the CAM and is used as a tag to the context information residing in local memory. The CAM holds the local memory index and a reference count. Processing is divided into two phases: a read or populate phase and a consume phase. In the read or populate phase, each thread of a microengine looks up the context identifier in the CAM and, in the case of a

miss, adds the identifier to the CAM. In the case of a miss, it also issues a read to the external memory unit for loading the context information associated with the context identifier to the local memory. In the case of a CAM hit, the thread simply increments the reference count of the CAM indicating its interest in the same data.

In the consume phase, threads access the data already available in the local memory and decrement the reference count. If the reference count of the CAM entry is zero, it is the last thread accessing this data; hence, it flushes this data to the SRAM unit. The benefit of this scheme is that

multiple packets refer to the same context information in local memory (instead of SRAM). Data is read only once from the external memory unit into the local memory. All subsequent modification of the data occurs in the local memory. Finally, one or more writes from local memory to external memory are performed depending on the CAM eviction policy. Figure 5 illustrates this approach in detail.

The hash engine or the CRC engine available in the IXP2400 can be used to generate the 32-bit unique context identifier given a 5-tuple from the RTP header.
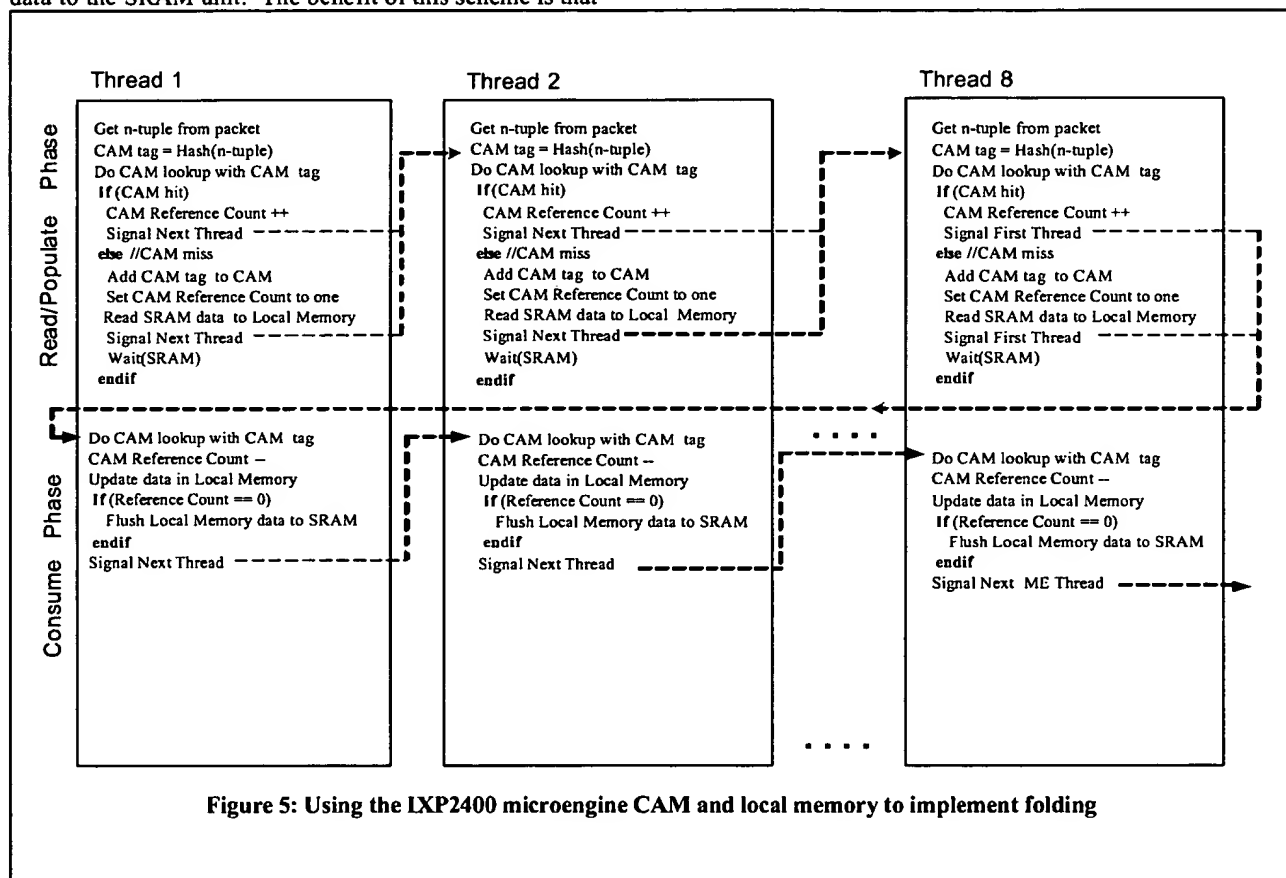

Figure 5: Using the IXP2400 microengine CAM and local memory to implement folding

Another technique to speed up processing is to cache the compressed/decompressed packet headers in the microengine local memory so that the compressor/decompressor need not access the packet header from the external memory unit.

IPv6 over IPv4 Tunneling (Interoperability)

Driven by a need for large numbers of uniquely addressable wireless devices, IP networks will gradually transition from a pure IPv4-based network to an IPv6-based network. The key to a successful transition is

interoperability with the large installed base of IPv4 hosts and routers. Maintaining interoperability with IPv4 while deploying IPv6 will streamline the task of transitioning the Internet to IPv6. This can be achieved using multiple mechanisms. The IPv6-IPv4 interoperability block (see Figure 4) uses IPv6 over IPv4 tunneling (encapsulating IPv6 packets within IPv4 headers to carry them over IPv4 routing infrastructure) to help realize this transition.

The main function of this block is to provide encapsulation and decapsulation of IPv6 datagrams. The

functionality includes removing and attaching appropriate IPv4 headers to IPv6 datagrams. Similar to link-classify and encapsulate blocks, the challenge is to be able to read or write layer-3 headers at arbitrary byte offsets. Once again, byte_align instruction, index mode addressing, and caching of protocol headers in local memory allow efficient implementation of these blocks.

This block integrates seamlessly with the IPv4 and IPv6 forwarding blocks. The IPv6 over IPv4 tunnels are configured by using special IPv6 and IPv4 route table entries. Hence the IPv6 and IPv4 forwarders are used to direct traffic transitioning from IPv6 to IPv4 networks and vice-versa to the interop block (see Figure 4). The IPv6 and IPv4 forwarders can also be configured to run independently (dual IP layer) on the same router. This dual IP layer functionality is also a requirement for successful transition from an IPv4 to IPv6 Internet.

Additional transition blocks can leverage the existing blocks to support advanced mechanisms like Network Address Translation-Protocol Translation (NAT-PT). Also, as the transition mechanisms are evolving, the programmable IXP2400 makes the task of adding software blocks supporting newer and more efficient transition mechanisms relatively painless. The possibility of being able to reprogram the IXP2400 in the field to support evolving networking standards is key to achieving rapid transition to the new standards.

THE QoS PROCESSING STAGE

The 3G wireless network defines four QoS classes: a low-delay conversational class for voice traffic, a constant delay streaming class for streaming video, a payload-preserving interactive class for web browsing, and a best-effort background class for e-mails and downloads. While the IXP2400 is fully programmable, the QoS software building blocks can be designed to be configurable to meet the needs of the 3G QoS classes. This section describes QoS building blocks on the IXP2400 and how they can be configured for 3G QoS.

The three functional blocks associated with QoS processing are the queue manager, the scheduler, and the rate shaper (Figure 6).

The queue manager block performs the packet enqueue and dequeue operations and updates and maintains queues. Fortunately, the SRAM Q-Array hardware-assist can be used for this purpose. The SRAM Q-Array hardware is used to cache the most recently used 64 queues in the SRAM controller. The Q-array data structure caches the head and tail pointers of the queue, as well as the number of entries currently present in that queue. Caching queue entries (in the memory controller) helps components like WRED and QoS, which manipulate large number of queues. The Q-Array hardware along with the CAM (local to each microengine) enables them to cache and access these queues efficiently.

Figure 6: QoS functional blocks
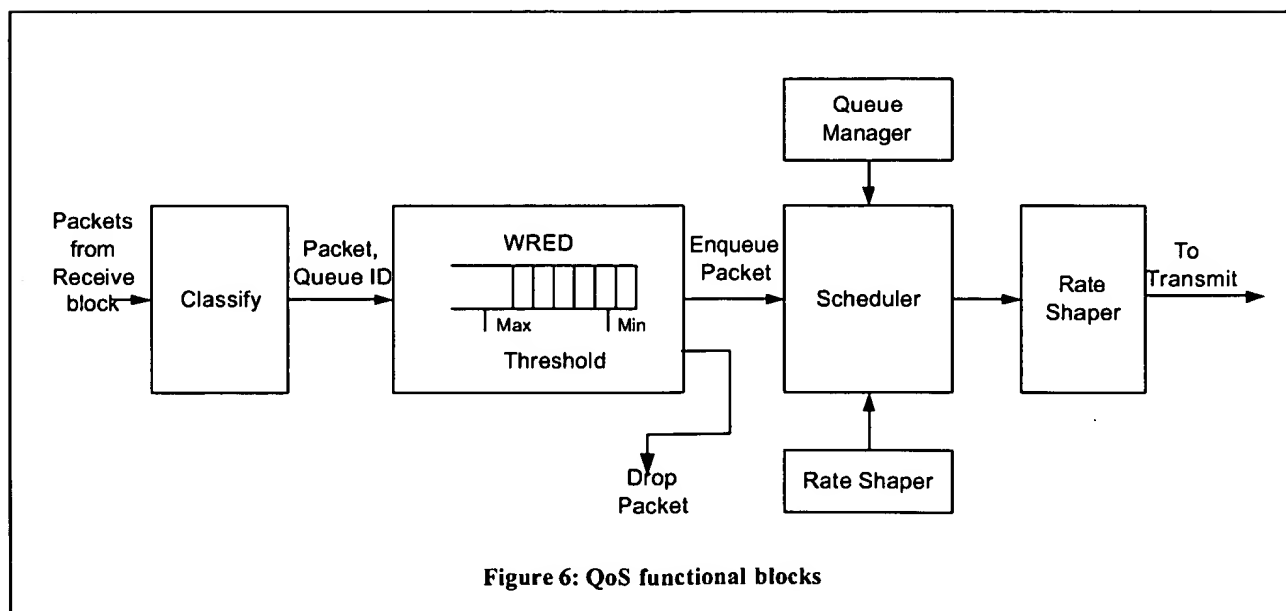
The queue manager receives requests to enqueue packets from the packet processing threads through the hardware-assisted producer consumer rings. Requests contain packet handle and queue number, to which packets must be enqueued. The queue manager also processes requests from the scheduler to dequeue packets.

In addition, the queue manager maintains the queue transition information that will be used by the scheduler. A queue transition occurs either when a packet is enqueued to an empty queue or when the queue is emptied out by dequeue of the last packet in a queue. Furthermore, the queue manager uses payload size information returned from dequeue operations to update credit information used for scheduling decisions and to update rate-shaping information.

The packet scheduler can be configured for Deficit Round Robin (DRR), Round Robin (RR), multi-level hierarchy (as in DiffServ), or any other proprietary scheduling scheme. Because the scheduler cannot tolerate any inherent delays, all its internal data structures such as queue status vectors, credit status vectors, shaping status vectors, and Round Robin masks are maintained in single-cycle access local memory.

The rate shaping is used to limit the rate at which packets are sent out on a virtual interface. This is needed when a single physical link aggregates one or more virtual links that are demultiplexed at the other end of the link. When a packet is scheduled to transmit on an interface, the shaper calculates the time slot at which packet transmission can be allowed the next time and registers that interface in the timing calendar queue. The rate shaper also turns the interface off to stop any further scheduling on this interface. When the timer reaches the appropriate time-slot entry in the timer calendar, it turns on the interface again. The microengine architecture provides a hardware timer capable of signalling at a period of 16 clock cycles. The current time (in ME cycles) is accessible in 3 cycles and can be used to accurately control the rate at which packets are sent out.

In addition, congestion avoidance mechanisms such as Weighted Random Early Detection (WRED) can also be implemented in the packet processing stage. The IXP2400 microengines use the well-known technique of using the Linear Feedback Shift Register (LFSR) hardware to generate extremely good 32-bit pseudo-random patterns. This hardware supplies the random-number to the microengines in two clock cycles. The pseudo random number generator is used in components such as WRED, which require an efficient random number generator to compute packet drop probability.

Each queue or interface on the system can be configured to have a different set of WRED thresholds, drop probabilities, and scheduler priorities. The rate shaper can also be configured to support a different transmit rate on each logical interface.

This ability to configure WRED, scheduler, and rate shaper allows the implementation of the 3G QoS classes. For instance, the low-delay conversational class traffic can be classified and placed into queues that have lower WRED thresholds and higher scheduler priorities. The lower WRED thresholds ensure lower latencies. The higher priority assigned to these queues ensures that the packets are transmitted ahead of others. On the other hand, best-effort traffic can be classified into queues with high WRED thresholds and placed in the best effort queue.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

CONCLUSION

The IXP2400, a programmable network processor with a comprehensive set of hardware assists (summarized below) is ideal for present as well as future needs of evolving all-IP 3G wireless networks. Even after implementing the key processing functions, sufficient headroom remains available for adding new services such as Multi Protocol Label Switching (MPLS), and the upgrades can be performed in the field. This allows nodes such as RNCs to grow and evolve with the upcoming roll out of wireless services.

Table 1 summarizes packet processing functions and the IXP2400 hardware feature that can be utilized to implement them efficiently.

Table 1: Packet processing functions

Processing Block	IXP2400 Hardware Assists
Link-layer classify and encapsulation/ decapsulation	Hardware-assisted scratch rings to dequeue packets from Rx. Single-cycle byte-align instructions and index-mode addressing to read layer-3 headers at arbitrary offset. Local memory to cache headers so that other blocks can use them.
Layer-3 forwarding	Hardware multi-threading to hide memory latencies. Caching packet headers and

Processing Block	IXP2400 Hardware Assists
	route-table data in local memory. Hardware-assisted scratch rings for sending exception packets to and from ME to XScale™.
Header compression/ decompression	CAM and local memory for caching. Hardware-assisted hashing for generating compression contexts.
QoS	SRAM Q-array for hardware-assisted enqueue and dequeue. CAM and local memory for caching the most active queues. Fine granularity timer to shape traffic. Local memory for real-time access to scheduling and shaping data structures.
WRED	Pseudo-random number generator to calculate drop probability.
IPv6 over IPv4 Tunneling	Local memory to cache layer-3 headers. Single-cycle byte-align instructions to assist encapsulation and decapsulation of IPv6 frames in IPv4 frames at arbitrary offsets. XScale to dynamically configure tunnels and routes.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

ACKNOWLEDGMENTS

The authors acknowledge the contributions of Uday Naik, Alok Kumar, Chen-Chi Kuo, Larry Huston, Philip J. Young, Sridhar Lakshmanamurthy, and Makaram Raghunandan.

REFERENCES

[1] Third-Generation Partnership Project, Technical Specification Group Services and System Aspects, QoS Concept and Architecture (Release 5), 3GPP TS 23.105 V5.1.0.

[2] Third-Generation Partnership Project, Technical Specification Group Radio Access network, IP Transport in UTRAN (Release 5), 3GPP TR 25.933 V5.0.0.

[3] "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links," *IETF RFC 2508*.

[4] Sally Floyd and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transaction on Networking*, August 1993.

AUTHORS' BIOGRAPHIES

Harsh Vipat is an applications engineer in the Network Processor Group. His interests include networking, operating systems and distributed systems. He has a Master's degree in Computer Science from Arizona State University. He can be reached via e-mail at harshawardhan.vipat@intel.com.

Manohar Ruben Castelino is an applications engineer in the Network Processor Group. He has worked in projects primarily in the networking and network management areas. His interests include networking and compiler design. He has a B. E. degree from KREC India. He can be reached at manohar.castelino@intel.com

Philip Mathew is a network application software engineer at Intel's Network Processor Division. His professional interests include computer networking, embedded programming and object-oriented programming. He received his Master's degree in Computer Applications from the University of Calicut, India, in 1994. He can be reached via e-mail at philip.mathew@intel.com.

Auro Tripathy is an applications engineering manager in the Network Processor Group. He has led projects in the wireless and broadband access aggregation areas. His interests include networking, digital video, and embedded developments tools and languages. He has a B. Tech degree from IIT Kharagpur, India, and an MSCS degree from Wayne State University, Detroit. He resides in Milpitas, California, and can be reached via e-mail at auro.tripathy@intel.com.

Implementing Voice over AAL2 on a Network Processor

Jaroslaw Sydir, Prashant Chandra, Alok Kumar,
Sridhar Lakshmanamurthy, Longsong Lin, Muthaiah Venkatachalam
Intel Communications Group, Intel Corporation

ABSTRACT

Programmable network processors are emerging as a versatile component for building telecommunications equipment because they can be programmed to perform a variety of different packet-processing functions, while allowing the equipment vendor to differentiate their products by supporting unique value-added features. A variety of applications with rather different characteristics and requirements can be deployed on a network processor. This ability to support a broad range of applications is one of the keys to the success of network processor products in the marketplace.

In this paper we describe a Voice over AAL2 (VoAAL2) processing application, as specified in International Telecommunications Union (ITU) recommendations I363.2 and I366.2. This application must satisfy the real-time requirements inherent in voice applications. It must use a jitter buffer and scheduler to remove jitter introduced in the network and a timer-based scheduler to ensure that voice packets do not incur too large of a processing delay. Also, because packets from low-data-rate voice channels are aggregated into high-data-rate Asynchronous Transfer Mode Virtual Circuits (ATM VCs) and vice versa, different components within the application must operate at different rates (some dealing with voice packets, others dealing with ATM cells). These requirements present some unique challenges to network processors that have traditionally been designed to support high-speed applications such as basic IP processing, where the processing of packets is very uniform and can be performed in a deterministically ordered pipeline. Applications such as VoAAL2, on the other hand, are a lot more asynchronous in nature. We discuss the requirements that applications like VoAAL2 place on network processor design and provide an example of how

a VoAAL2 application can be implemented on the IXP2400 processor.

INTRODUCTION

Programmable Network Processors (NPUs) offer telecommunications equipment manufacturers a flexible platform for building a variety of different equipment. The power of NPUs is that they can be programmed to perform many different packet-processing functions to support a variety of different protocols and standards. This flexibility allows equipment manufacturers to utilize the same NPU or family of NPUs across different product lines. It also allows them to easily evolve their products to support evolving standards and to provide unique value-added features within these products.

Until recently, most network processors have been designed mainly to perform basic IP packet processing, as described in RFC 1812 [5], at very high line rates. This basic IP packet processing is fairly simple and deterministic. All packets are subject to roughly the same processing. Quality of Service (QoS) guarantees are either not provided at all or are provided on a course, per-class granularity with no guarantees on packet delay or packet delay variation (jitter).

Unfortunately, real-world packet-processing applications are much more complex and diverse than this basic IP processing application. IP processing performed by today's routers includes many additional features such as DiffServ QoS, policy-based routing, and packet filtering that make the packet processing applications much more complex and less uniform. Other packet processing applications deal with real-time traffic and therefore have stricter real-time processing requirements.

In order to provide the level of flexibility sought by equipment manufacturers, an NPU must be able to support diverse packet-processing applications dealing with

connectionless and connection-oriented protocols with a variety of quality of service models and requirements. Different applications require different programming models, which must be supported by an NPU.

An example of an application that presents a different set of requirements than the basic IP forwarding application is the Voice over AAL2 (VoAAL2) application. In this paper we discuss the special requirements and challenges presented by the VoAAL2 application. We describe the architecture and design of a VoAAL2 application that we have developed for the Intel IXP2400 processor and discuss the features of an NPU that are required to support this type of application.

IXP2400 NETWORK PROCESSOR

The IXP2400 is a next-generation network processor developed by Intel Corporation. It is fully programmable, offering a very flexible programming model and support for a broad range of diverse packet processing applications. In this section we highlight some of the IXP2400 features that are utilized by the Voice over AAL2 (VoAAL2) application. "Network Processor Performance Analysis Methodology," by Sridhar Lakshmanamurthy, et. al, provides a complete overview of the IXP2400 architecture [1].

The IXP2400 is a multi-threaded multi-processor system. Packets enter the IXP2400 through a configurable industry standard interface that supports Packet over SONET and UTOPIA interfaces. Packet processing is performed by eight packet-processing engines called MicroEngines (MEs). Each ME has eight hardware execution contexts (alternately referred to as threads). Each context has its own register set, so that swapping between them is a very fast (one instruction cycle) operation. The MEs use a non-preemptive context scheduling model, where the swapping out of contexts occurs under software control, and ready-to-run contexts are scheduled using a Round Robin scheduling discipline.

Each ME contain an Arithmetic Logic Unit (ALU) and a Content Associated Memory (CAM) unit, which allows the application to compare a key value to the keys of all CAM entries in one instruction cycle. The MEs also provide byte-alignment support to allow applications to manipulate packet headers and data that are not always aligned on four-byte boundaries. Each ME contains 640 longwords of local memory, where packet headers can be stored temporarily while they are processed. Finally, the

MEs provide real-time timers, which allow a thread to specify a time in the future when it should be awakened.

The IXP2400 contains interfaces to external SRAM and SDRAM memories.

VOICE SERVICE REQUIREMENTS

Transmitting voice signals over a network places some specific real-time requirements on the network. Voice (for example, a telephone call) is sampled at periodic intervals, and those samples are transmitted across the network and replayed at the other end at the same rate. Each voice sample is subject to a transmission and propagation delay as it traverses the network. This delay cannot be too large if the conversation is to flow at a normal pace. More importantly, variations in the delay experienced by different samples (referred to as jitter) cannot cause the replay of those samples to occur at a variable rate, or the quality of the voice experience by the listener will degrade.

Traditional voice networks, Time Division Multiplexing (TDM) networks, solve this problem by reserving the capacity for the voice samples of a call at the time the call is set up and synchronizing the transmission of voice samples throughout the network. This provides a very predictable environment for voice applications and guarantees that the jitter and delay experienced by voice samples is within specified limits and yields the voice quality that we are used to when using the telephone. The drawback of this approach is that a 64Kb/s channel is reserved for the duration of the voice call and cannot be used to carry voice samples from other calls even during periods of silence.

Transmitting voice over packet networks, such as ATM, can solve this resource usage efficiency problem because voice samples from different calls are allowed to use the bandwidth from a call during periods of silence. The challenge is to allow this type of bandwidth sharing while still providing the same delay and jitter guarantees in order to maintain the same level of voice quality as a traditional voice network.

VoAAL2 SERVICE

Typical VoAAL2 Deployment

Figure 1 illustrates the typical configuration of a Voice over AAL2 (VoAAL2) service in a network. Voice calls originate on the Time Division Multiplexing (TDM) network.
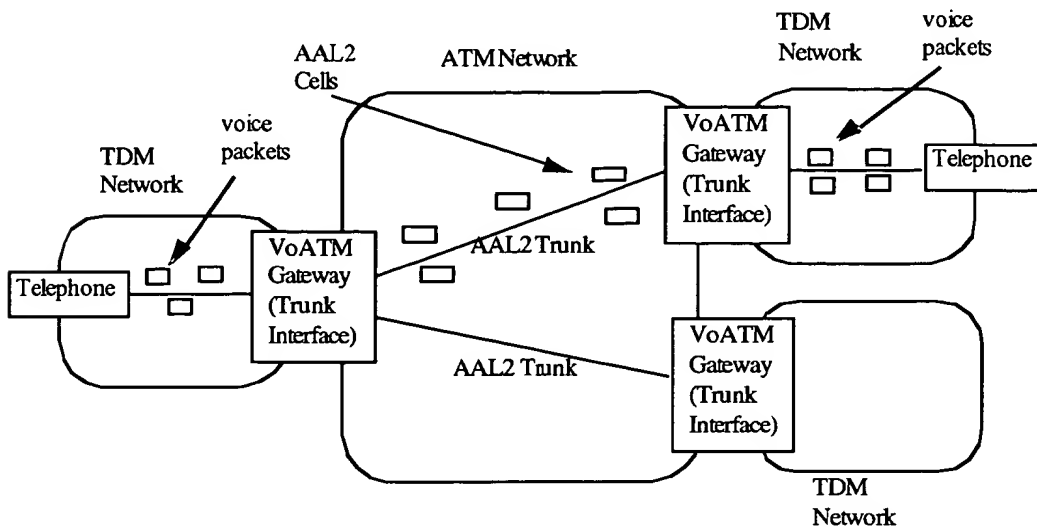
Figure 1: Configuration of a Voice over AAL2 (VoAAL2) service

The analog voice signal is sampled and digitized in the VoATM gateway to produce a stream of voice packets. (Note: TDM networks also carry digitized voice and other traffic.) These voice packets are transported across the ATM network in AAL2 trunks. An AAL2 trunk is an ATM Virtual Circuit (VC) used to transport AAL2 traffic. At the far end of the ATM network, the voice packets are converted back to an analog signal and transmitted over an analog voice network.

Figure 2 illustrates the relationship between voice calls, AAL2 channels, AAL2 Trunks, and ATM VCs. There are many Digital Signal Processors (DSPs) within a VoATM Gateway. Each DSP processes a given number of voice

calls. Each voice call is mapped to an AAL2 channel within an AAL2 trunk. There are 256 AAL2 channels within each AAL2 trunk. The AAL2 channel is identified by the Channel Identifier (CID). There are many AAL2 trunks in the system. Different voice calls from a given DSP can correspond to AAL2 channels within the same or different AAL2 trunks. The inverse relationship holds at the far end of the ATM network, where packets from each AAL2 channel are transformed into voice packets destined for a specific DSP. The VoAAL2 application transforms voice packets to AAL2 packets and transmits them on the correct AAL2 channel. At the other end of the ATM network, the VoAAL2 application performs the inverse operation.
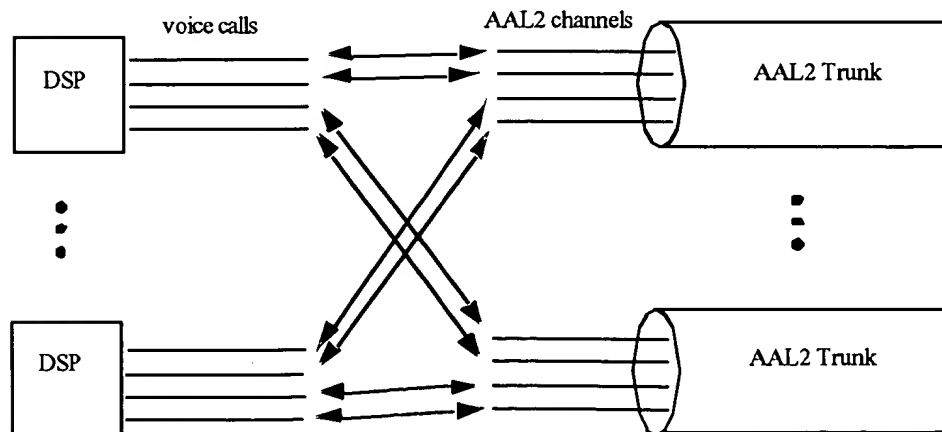
Figure 2: Relationship between voice calls and AAL2 channels

AAL2 Standards

The Voice over VoAAL2 service is specified by the International Telecommunications Union (ITU). AAL type 2 is subdivided into the Common Part Sublayer (CPS) and the Service-Specific Convergence Sublayer (SSCS). Recommendation I.363.2 specifies the CPS layer for all AAL type 2 applications [2]. This layer defines a packet format with a three-byte packet header, which contains the length of the packet, a User-to-User Indication (UUI) field, whose content is specified by the layer above, and a header error correction field. Recommendation I.366.2 specifies an SSCS layer for trunking of traffic from narrow band networks (ISDN or analog networks) over AAL2 [3]. I.366.2 defines a number of services for transporting audio and data traffic, and signaling over an AAL2 network. For the audio service, the SSCS layer does not define its own header. It simply specifies the format and values that are passed to the CPS layer and transmitted in the UUI field of the CPS header.

PACKET PROCESSING IN THE VoAAL2 APPLICATION

Figure 3 illustrates the relationship between the different types of packets. In the Digital Signal Processor (DSP) to Asynchronous Transfer Mode (ATM) direction, digitized voice packets are received from the DSP chip. Each voice packet is mapped to one Service-Specific Convergence Sublayer (SSCS)/Common Part Sublayer (CPS) packet. Multiple CPS packets are multiplexed within an AAL2 cell.
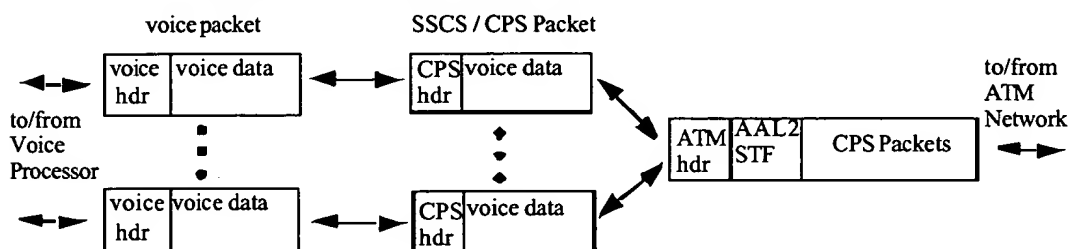

Figure 3: Voice over AAL2 packet transformations

The voice packets contain a header that indicates the identity of the voice channel, the length of the packet, and the encoding algorithm that was used. The voice header is a software convention established between the Voice over AAL2 (VoAAL2) application and the DSP and is not part of the standards. The information contained in this header could also be communicated by some out-of-band communication mechanism.

The application looks up the AAL2 channel, within a specific ATM Virtual Circuit (VC), that is used to transport this call. The DSP header is stripped off of the voice packet and SSCS processing is performed. SSCS processing involves generating the proper sequence number that is carried in the User-to-User Indication (UUI) field of the CPS header. This sequence number is maintained on a per-voice-call basis.

Next, CPS processing is performed. This entails the creation of the CPS header and the generation of a CRC-5-based header error correction field. Multiple CPS packets can be packed into the payload of an ATM cell, and the contents of a given CPS packet can be split across successive ATM cells (within the same VC). The first byte of AAL2 ATM cells contains a field called the Start Field (STF), which indicates the length of the data and offset to the start of the first CPS packet within the cell.

CPS packets destined for the same VC are accumulated until either a cell is completely filled or until the timer_CU has expired. Recommendation I.363.2 specifies the use of the timer_CU to force a cell to be sent after a certain amount of time, even if it is not full. The time_CU is used to make certain that the processing delay incurred by a CPS packet (voice packet) is less than a specified number. A timer_CU is maintained for each VC. When the first CPS packet is received (for a cell on that VC) the timer_CU is started for that VC. When additional CPS packets are received and the cell is filled, the timer_CU is canceled (or restarted if there is enough data to start a new cell). If the timer expires before the cell is full, the empty part of the cell is padded with zeros and the cell is sent even though it is not full.

The processing in the ATM to DSP direction is the inverse of the processing in the DSP to ATM direction. A cell is received on a particular VC. Within the cell are one or more CPS packets. The first packet may be a partial packet, part of which may have come in the previous cell on the same VC. Also, the last CPS packet may not be complete. The CPS packets within a cell can be destined for the same AAL2 channel or different channels.

The CPS packets are extracted from the ATM cell and reconstructed, and the CPS headers are verified using the CRC5 value in the header. For each packet the voice channel to which it belongs is determined as a function of the VC and AAL2 CID. The SSCS sequence number is also verified, and the packet is discarded if it is corrupted or misordered. The DSP header is prepended to the payload of the SSCS packet to create a DSP packet.

Each voice packet encodes a specified time interval of the voice signal. The timestamp for a packet represents the beginning of this interval. The SSCS sequence number captures the time of a packet relative to the time of the previous packet. The timestamp for packet n (Tn) is given by the formula: $Tn = Tn\text{-}1 + ((Sn - Sn\text{-}1)*I)$, where $Tn\text{-}1$ is the timestamp of packet $n\text{-}1$, Sn and $Sn\text{-}1$ are the sequence numbers of packets n and $n\text{-}1$ respectively, and I is the interval length for this call.

When a packet is received, its SSCS sequence number is used to generate a timestamp, which is used to perform jitter removal from the stream of voice packets that make up a call. Jitter is the variable inter-packet gap caused by network queuing and transmission delays experienced by successive packets or cells from one connection. Network jitter causes voice packets from a channel to be either bunched together or spread out in time, thereby making the inter-packet gap smaller than or greater than the codec sampling interval. Before the voice packets can be played out to the listener or transmitted over a TDM link, this variability in the inter-packet gap must be removed. Removing jitter requires collecting enough voice packets from a channel in a buffer so that the voice packets can be played out with a constant inter-packet gap corresponding to the codec interval. This dejittering operation must be performed individually for each voice call.

QoS CONSIDERATIONS

Traffic management in Asynchronous Transfer Mode (ATM) networks is specified by the ATM forum in the Traffic Management Specification [4]. The TM4.1 specification defines six service categories that are used to provide different levels of QoS guarantees to different types of traffic. Each service category is defined in terms of the characteristics of the traffic that can be afforded this service (called the traffic contract) and the types of QoS guarantees that traffic which conforms to the traffic contract will receive.

The Constant Bit Rate (CBR) service category provides a service similar to that provided by a TDM network. CBR traffic is characterized by a peak cell rate. A conformant CBR traffic stream cannot exceed its peak rate or a maximum cell delay variation. The traffic is guaranteed very low losses and a maximum cell transfer delay. Voice and circuit emulation services are potential users of this service.

The Real-Time Variable Bit Rate (rtVBR) service category provides loss and delay guarantees to traffic whose bit rate is variable. The traffic is characterized by a peak rate, a sustainable rate, and a maximum burst size. A conformant traffic stream must not exceed its sustainable rate over long timescales; it can burst at rates up to its peak rate, up to its maximum burst size. The traffic is guaranteed very low losses and a maximum cell transfer delay. Real-time applications such as voice and video are potential users of this service.

The Non-Real-Time Variable Bit Rate (nrtVBR) service category is intended for non-real-time applications with a bursty traffic pattern. The traffic and conformance criteria for this service are characterized in the same way for the rtVBR service category. The network offers a loss guarantee, but no packet delay guarantees.

The Unspecified Bit Rate (UBR), Available Bit Rate (ABR), and Guaranteed Frame Rate (GFR) service categories are intended for that transport of data traffic. UBR is a best-effort service, where no restrictions are placed on the traffic and no guarantees are provided by the network. ABR provides a loss guarantee and utilizes closed loop feedback control to throttle the traffic sources in order to avoid losses in the network. Finally, GFR is intended to provide a service similar to that offered by frame relay for IP applications.

The TM4.1 specification describes a number of mechanisms for implementing traffic management within the network. Call Admission Control (CAC) is used to determine whether the network has the resources to support a connection and to reserve these resources for the connection. Policing is performed at the edges of the network to make certain that the traffic entering the network conforms to its traffic contract. Shaping is used to transform a traffic stream into one that meets a different traffic contract. Finally, scheduling is used to ensure that the resources reserved by a connection are made available to the cells traversing that connection.

From a traffic management perspective the VoAAL2 application is a user of the network. AAL2 trunks (VCs) are generally established as CBR or rtVBR connections, and the traffic stream produced by the VoAAL2 application for each VC must conform to the traffic contract for that VC. TM4.1 uses the Generic Cell Rate Algorithm (GCRA) for defining the conformance of a traffic stream to its traffic contract. GCRA has two parameters T and τ. T is the inverse of the rate allocated to the flow by the network. The rate here could mean either

peak rate or average rate, depending upon the service class. The second parameter τ represents the deviation from the theoretical arrival times of the cells in a flow that can be tolerated by the network. The algorithm maintains the theoretical earliest arrival time for the next cell. When a cell arrives, its actual arrival time is compared to the theoretical arrival time. If the cell has arrived later than the theoretical earliest arrival time or less than τ units of time earlier than this time, then the packet conforms. Otherwise, it does not. The theoretical arrival time is calculated as a function of the actual arrival time of the current cell and the parameter T.

VoAAL2 APPLICATION ON IXP2400

We have implemented the Voice over AAL2 (VoAAL2) application on the Intel IXP2400 processor. In this section we describe the design of this application and discuss some of the challenges involved.

DSP to ATM Processing Design

Figure 4 illustrates the major components, data structures and control and data flow for the Digital Signal Processor (DSP) to Asynchronous Transfer Mode (ATM) direction. Thin dotted lines within the figure represent a relationship between data items. Solid lines represent data flow, and thick dashed lines represent control flow.
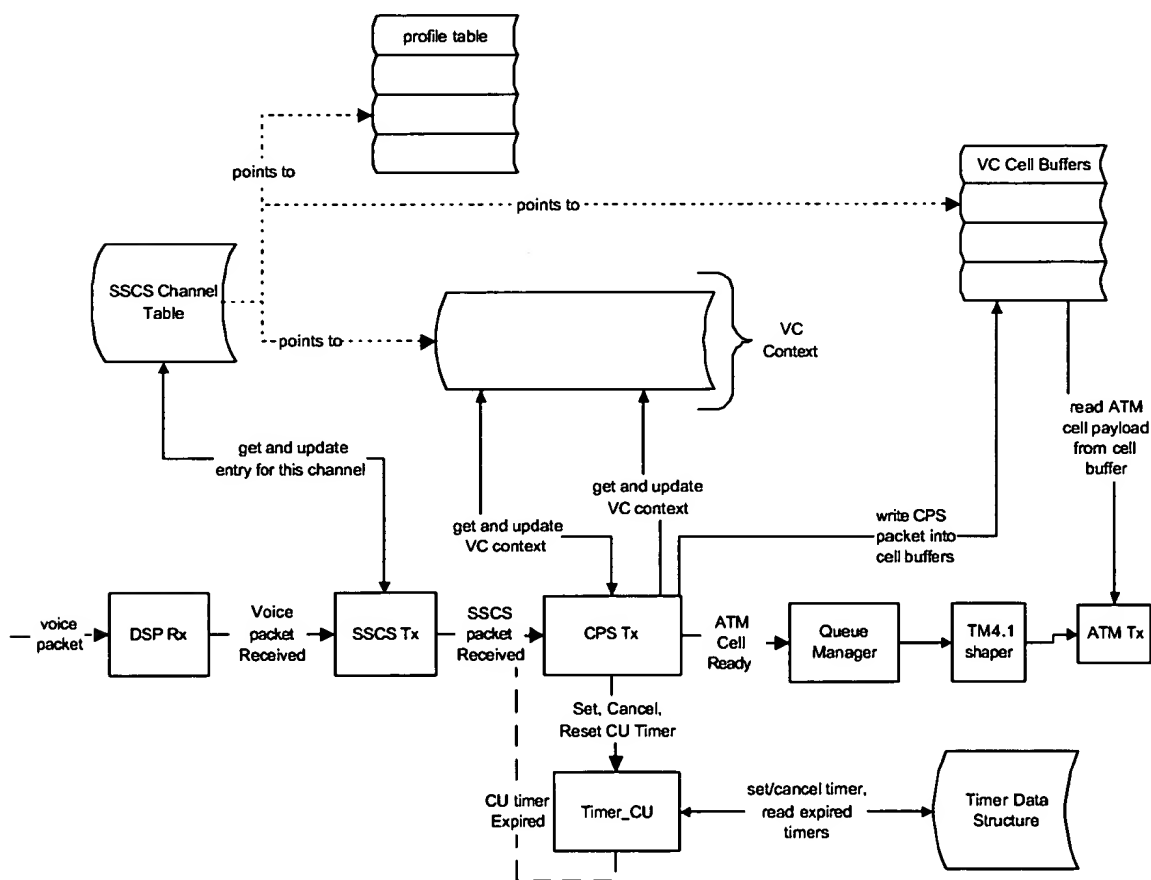

Figure 4: DSP to ATM processing

The **DSP Rx** component receives voice packets from the DSPs. It reads the header that is created by the DSP and extracts the relevant meta-data that describes the voice packet. It then places this meta-data into a message to the SSCS Tx component. It is important to separate out the processing of voice packets from the voice processor because in other versions of this application, the voice packets may come from a different source, such as another AAL2 connection (AAL2 switching), or from a voice-over-IP connection.

The SSCS Tx component receives messages from the DSP Rx component, indicating that a voice packet is ready to be processed and performs SSCS processing to produce the value of the User-to-User Indication (UUI) and length fields that go into the Common Part Sublayer (CPS) header. It accesses two main data structures in performing this processing. First, it looks up the entry in the **SSCS channel table** that corresponds to the AAL2 channel over which the voice packet is to be transported. The SSCS channel table contains an entry for each SSCS channel. Each entry contains the Channel Identifier (CID) of this channel, the Virtual Circuit (VC) within which it exists, the profile for this call, and the SSCS sequence number. Next, the SSCS Tx component searches the profile table that describes the profile used for this channel for the entry that correspond to this voice packet. The **profile tables** are read-only tables that describe the UUI field encoding, sequence number interval, and length for each audio encoding algorithm that can be used within the profile. Within a profile table there is an entry for each encoding algorithm supported in the profile. Each voice packet must match one of the entries with the profile. A profile table exists for every profile supported by the application.

When it had completed the SSCS processing, the SSCS Tx component sends a message to the CPS Tx component indicating that a SSCS packet has been received.

The **CPS Tx** component encapsulates SSCS packets in CPS packets and packs CPS packets into AAL2 cells. It is driven by the arrival of two types of messages. Messages indicating that an SSCS packet has been received are sent by the SSCS Tx component. In response to these messages, this component gets the VC context that corresponds to the VC on which the packet has arrived. The **VC context** stores the state of an AAL2 VC. It contains information that is required to determine if the timer for a VC should be set, canceled, reset, or left alone, and information about the cell buffer in which the current cell is being assembled. The CPS Tx component first performs the bookkeeping in order to determine what should be done with the timer_CU for this VC, and sends a message to the timer_CU component indicating the required action. The timer_CU can be set, canceled, reset, or left alone, depending on whether it was previously set and whether the data from the new packet has partially filled a new cell. The CPS Tx component then creates the CPS header to produce a CPS packet that contains an SSCS packet, which contains the voice packet. The CPS packet is written into the current cell buffer immediately following the previous CPS packet destined for this VC. The packet may not entirely fit into the current cell, in

which case the part that fits into the current cell is written there and a message is sent to the Queue manager component, indicating that the cell is ready to be sent. The remainder of the CPS packet is written to the next cell buffer. (A maximum size CPS packet can fill up two ATM cells.) The VC context is updated and written back to memory.

Messages indicating that the timer_CU for a VC has expired are sent to the CPS Tx component by the timer_CU component. In response to these messages the CPS Tx component gets the VC context that corresponds to the VC whose timer_CU has expired. It determines how many bytes of padding must be written to complete the cell, writes this padding to the cell buffer, and sends a message to the queue manager component indicating that the cell is ready to be sent. Finally, it updates the VC context to account for the actions that were taken.

The **timer_CU** component implements the timer_CU functionality. It accepts requests to set, cancel, and reset the timer for specific VCs. It is also responsible for firing the individual timer_CUs that are set (and canceled) for individual VCs. The **timer_CU structure** is a calendar queue data structure used to store the timer_CU entries for active VCs. The timers are stored in buckets, and each bucket is associated with a time interval. The timer_CU component wakes up at the end of each time interval and sends timer_CU-expired messages to the CPS Tx component for each VC that had a timer_CU set to go off during the previous time interval.

The **queue manager** component manages a set of queues in SRAM. There is a queue for each ATM VC. Cells are placed into the queue by the CPS Tx component and removed by the TM4.1 shaper component.

The **TM4.1 shaper/scheduler** component consists of three blocks. The TM4.1 shaper block receives input from the queue manager when a cell from a particular VC Queue (VCQ) has been dequeued and there are cells remaining in that VCQ (this is called *cell dequeue without transition*) or when cells are enqueued into an empty VCQ (this is called *enqueue with transition*). The shaper computes the *earliest* departure time for the cell using the Generic Cell Rate Algorithm (GCRA) traffic descriptors. It passes on the *earliest* departure time, the VCQ number, and the service category for the cell onto the TM4.1 writeout block.

The design uses time queues to achieve compliance and provide TM4.1 functionality. Time queues are depicted in Figure 5. The time axis can be divided into small units of cell transmission slots. In each slot, one or no cells depart.
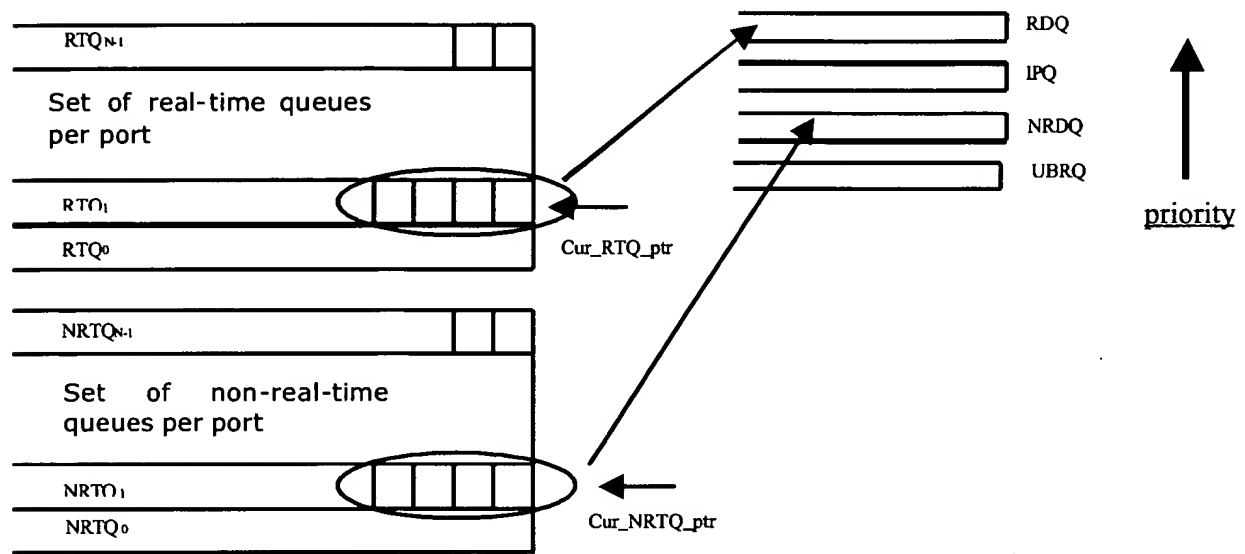
Figure 5: TM4.1 scheduler time queues

A time queue is the aggregation of several cell transmission slots and hence represents an interval of time. The time queue holds the cells that are meant to be transmitted during this time interval. There are a fixed number of time queues in the system (that can be derived based on the link rate), the slowest VC bit rate, and the aggregation level of the time queue. The sum total of all the time intervals represented by the all-time queues would constitute the time horizon. The time horizon is nothing but the time after which the time queues wrap around. There are two sets of time queues: one for real-time traffic (called the *real-time time queue*), such as CBR and rt-VBR, and the other for non-real-time traffic (called the *non-real-time time queue*), such as nrt-VBR and GFR.

The TM4.1 writeout block computes the time queue into which the cell needs to be written based on the *earliest* departure time. Once the time queue is computed for the cell, it writes out the cell into the real-time time queue if the traffic is CBR or rt-VBR and the non-real-time time-queue if the traffic is nrt-VBR. If no space is available in the time queues, the writeout block writes into a different data structure called the Intermediate Priority Queue (IPQ).

The scheduler block schedules out cells from the time queues, IPQ, and the UBR queues. It is essentially a priority scheduler with the highest priority for the real-time time queue, the next priority for IPQ, next for non-real-time time-queue, and the lowest priority for UBR.

When scheduling from a time queue, the scheduler is always in sync with or behind the real time. The design ensures that cells are not scheduled ahead of real-time, since this would violate the traffic contracts of the VC and create unfairness in the system.

Finally, the **ATM Tx** component performs the ATM header processing and transmits the cell.

ATM to DSP Design

Figure 6 illustrates the major components, data structures, and control and data flow in the ATM to DSP direction. The **ATM and CPS Rx** component receives ATM cells. When a cell is received, the ATM and CPS Rx component determine the VC to which this cell belongs. The **VC context** for this VC is then read in from SRAM. Since it is possible that a CPS packet was split across cells, this context contains information about such a split CPS packet. The ATM and CPS Rx component reads the contents of the ATM cell into the microengine and steps through the CPS packets contained within. It copies each CPS packet into a packet buffer, maps the Virtual Path Indicator (VPI), Virtual Circuit Indicator (VCI), and CID fields to the channel id, and queues each packet buffer for the SSCS Rx component to process. If a CPS packet is split across ATM cells, the ATM and CPS Rx component stores the reassembly context in the VC context, in order to allow the packet to be completed when the next cell for this VC arrives.
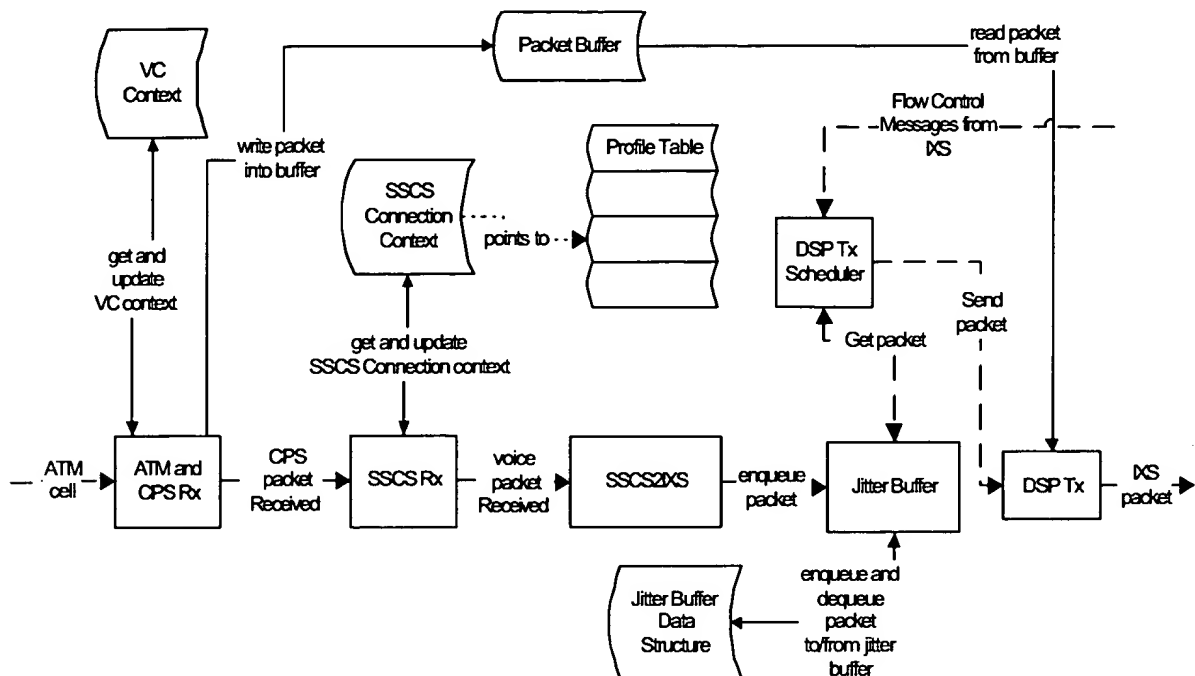
Figure 6: ATM to DSP processing

The **SSCS Rx** component performs the SSCS processing. It uses the channel id to access the **SSCS connection context** for this channel. The context contains the sequence number of the previously received SSCS packet and a pointer to the table that describes the profile for this connection. The structure of the profile tables was described in the previous section. The SSCS Rx component accesses the entry in the profile table that corresponds to the UUI field and length of the SSCS packet (in the CPS header). This entry specifies the encoding algorithm that was used, along with information used to determine the expected sequence number and corresponding timestamp for this packet. SSCS Rx checks the sequence number against the one received in the packet and generates the timestamp. If there are no errors, the packet is passed to the SSCS2DSP component.

The **SSCS2DSP** component creates a DSP packet header from the information that the ATM and CPS Rx and SSCS Rx components have extracted from the packet and profile table. It then passes the packet to the jitter buffer component.

The **jitter buffer** component enqueues the packet into a per-channel queue. The purpose of the Jitter Buffer component is to eliminate some of the jitter introduced into the voice packet stream in the ATM network. It does this by placing packets into proper time sequential order, applying a specified jitter delay, and playing them back at the proper rate (with jitter removed).

The **DSP Tx scheduler** component is responsible for scheduling the transmission of DSP packets to the DSP. This component registers itself with the MSF in order to receive flow control messages from the DSP. In each flow control message, the DSP indicates one or more channels, on which it is ready to receive a packet. For each such channel, the DSP Tx scheduler component asks the jitter buffer component to dequeue a packet from the jitter buffer. The jitter buffer returns either a buffer handle or an error. If the jitter buffer returns a buffer handle, the DSP Tx scheduler component passes this handle to the DSP Tx component. If the jitter buffer component returns an error, the DSP Tx scheduler component creates a DSP packet that indicates silence (an SID) and passes it to the DSP Tx.

The jitter buffer component receives requests from the DSP Tx scheduler to dequeue a packet on a specific channel. It determines if a packet is queued and ready to send for that channel and responds with the buffer handle of that packet or an error.

Finally, the **DSP Tx** component transmits voice packets to the voice processors.

CHALLENGES AND LESSONS LEARNED

In this section we discuss some of the challenges that must be surmounted in developing a Voice over AAL2 (VoAAL2 application) on a Network Processor Unit (NPU).

Processing Asynchronous Inputs within Many Contexts

The VoAAL2 application must support a large number of AAL2 Virtual Circuits (VCs). There is a requirement that within each VC, cells/packets be processed in the order in which they were received. (In the DSP to ATM direction, voice packets destined for a VC should be processed in order, while in the ATM to DSP direction, ATM cells received on a VC must be processed in order.) The data rates within VCs can be fairly small, so at any given moment the application is holding/processing cells/packets from a small subset of this total number of VCs. Because there are many VCs, the packets that the application is processing/holding at a given time are most likely all from different VCs, although this is not guaranteed and cannot be assumed by the application. The challenge is how to serialize the processing of cells/packets within each VC, while allowing cells/packets from different VCs to be processed in parallel.

Another problem is that the CPS Tx component must process voice packets received by the system as well as react to the expiration of the timer_CU. Timer_CU expiration events are not regular or predictable, since they are a function of the traffic patterns on individual VCs. The amount of processing required to process a packet that has arrived is much larger than that required to react to the expiration of the timer_CU.

We solve the problem of having to serialize the processing of packets/cells within each VC by dynamically binding VCs to threads. A thread receives a packet or message, determines which VC it belongs to, checks to see if any other thread is already processing packets/messages for that VC, and locks the VC if it is not already locked by another thread. The thread then processes the packet or message. When it has completed processing the packet or message, it checks to see if any other packets or messages have been queued for it to process (associated with this same VC). The thread processes any packets or messages that have been queued, and when there is none left, it unlocks the VC. On the other hand, if another thread has already locked the VC, the packet or message is queued for this other thread to process.

The process of locking a VC, unlocking a VC, and checking to determine if a VC is locked must be performed in an atomic fashion in order to ensure that two threads do not lock the same VC. In our design the entire component is implemented within a single ME, so we use a built-in CAM for storing the identity of the VC that is locked by each thread, allowing the operations of locking, unlocking, and checking to see if a VC is locked to be performed in one operation.

We found that the IXP2400 provides good support for the asynchronous programming model used in the VoAAL2 application. Central to this support are the CAM and local memory that are included in each ME. The IXP2400 also provides the basic support required to distributed this type of processing across multiple MEs. It provides atomic test and set operations in the shared SRAM, which can be used to implement locks. However, SRAM operations have a fairly large latency, making it difficult to use this mechanism for locking in high-performance applications. Additional hardware support for performing distributed locking from threads on multiple MEs would make it easier to implement such multi-ME asynchronous applications. On the other hand, it is generally possible to partition an application into components in such a way as to avoid asynchronous components that run on more than one ME.

Bit- and Byte-Level Memory Access

The CPS header and AAL2 cell are tightly packed structures, where fields are not aligned on four-, eight-, or even one-byte boundaries. This means that the VoAAL2 application must read and write from/to arbitrary bit and byte addresses as it creates/parses CPS packet headers and packs/unpacks CPS packets from AAL2 cells. This presents a challenge for any processor because memory systems generally support reads/writes of four- or eight-byte chunks of data, addressed on four- or eight-byte boundaries.

Our implementation utilizes specialized byte alignment hardware of the IXP2400 processor to merge and align the CPS packets as we pack/unpack them into/from AAL2 cells. Bit fields are accessed utilizing mask and shift operations provided by the Arithmetic Logic Unit (ALU). Efficient support for such data access is critical to support applications such as VoAAL2, where packets are small and protocol overhead must be minimized.

The problem of having to access unaligned data is solved by some combination of providing specialized hardware instructions and simply providing sufficient processor speed to allow the applications to perform the required data manipulations within the required time budget. We found that the IXP2400 provides a reasonable combination

of processing speed and specialized instructions to support this application.

Jitter Buffer

The purpose of the jitter buffer is to receive voice packets, place them in proper time sequential order, provide a specified jitter delay, and then present them for transmission. The jitter buffer must be large enough so that the slowest packets can arrive in time to be played out in the correct sequence. On the other hand, the jitter buffer must be small enough such that the delay introduced is minimized. In order to address these conflicting requirements, the jitter buffer can be dynamically resized based on measurements of actual network jitter. On lightly loaded paths, this allows for a minimum jitter delay and a higher quality of speech with less noticeable turnaround delay. On congested paths, the jitter delay can be increased so that fewer packets are missed or dropped due to the irregularity of their timing but with a more noticeable turnaround delay.

Implementing a jitter buffer offers some new challenges when compared to traditional First In First Out (FIFO) queues. The jitter buffer is a sorted queue based on the timestamps of arriving voice packets. Therefore, packets can be inserted in the middle of the jitter buffer. Packets can be dropped from a jitter buffer for two reasons: 1) the buffer is full; or 2) the packet is received too late. When packets are dropped because the queue is full, they are dropped from the front of the queue (packets with the oldest timestamp are dropped). Because the queue is allowed to contain packets representing a fixed time interval (the jitter delay value), the arrival of one voice packet may cause multiple older voice packets to be dropped if the time difference between the newest packet's timestamp and the oldest packet's timestamp is greater than the jitter delay value. Packets with duplicate timestamps are dropped. A further challenge is that a separate queue must be maintained for each of the many thousands of voice channels that are handled by the application.

Our implementation uses circular queues to implement the jitter buffer. Each circular queue has pointers to the packets with the oldest and newest timestamps. The position into which a new packet is inserted is a function of the difference between the packet's timestamp and the oldest packet's timestamp, along with the codec interval. To calculate this position we need to divide the timestamp difference by the coded interval. We implement this using fast reciprocal multiplication utilizing the multiplier of the IXP2400 MEs. Once the position is calculated, the insertion of the packet into the jitter buffer is the same as an insert into an array of the order $O(1)$.

The circular queues may have "holes," positions with no packets. When a packet must be removed from the jitter buffer, it is necessary to quickly skip over the holes to get to the position with a valid packet. We implement this search for a valid packet in $O(1)$ time by making use of the Find First Bit Set (FFS) instruction provided by the IXP2400 MEs. By maintaining a bit-mask of positions in the circular queue with valid packets, and using the FFS instruction, we can remove the next valid packet from the jitter buffer in constant time.

In summary, the jitter buffer implementation takes advantage of the hardware features provided by the IXP2400 network processor to implement, insert, and remove operations in $O(1)$ time. This allows for an efficient jitter buffer implementation that scales to a large number of voice channels.

TM4.1 Real-Time Scheduler

There are many challenges that must be overcome in developing a TM4.1-compliant scheduler. TM4.1 requires per-VC shaping and scheduling, and the number of VCs can be very large. Also, the implementation must scale with increases in line rate, as well as numbers of VCs.

The most interesting challenge is in providing the real-time scheduling required to support the CBR and rtVBR service classes. When servicing CBR and rtVBR traffic, the packet scheduler must transmit each cell within a certain time window in order for it to conform to the traffic contract.

Because the IXP2400 performs non-preemptive Round Robin scheduling, it is difficult for the software to perform such real-time scheduling. When a thread sets a timer and goes to sleep, expecting to be awakened when the timer has expired, it cannot be guaranteed that it will get awakened the instant that the timer expires, because another thread might be executing at the time that the timer expires, and other threads might be ahead of this thread in the Round Robin schedule. We found that TM4.1 scheduling is still possible, although the software must be carefully tuned to place an upper bound on the time between the expiration of a timer and the time that the thread is awakened, and the schedule must take this bound into account when setting its timers.

CONCLUSION

The flexibility and programmability of next-generation Network Processor Units (NPUs) will make them a key component of next-generation telecommunications equipment. NPUs can support a variety of packet-processing applications, with a variety of different requirements. The Voice over AAL2 (VoAAL2) application that we discussed in this paper presents a

number of challenges. We have demonstrated that these challenges can be overcome and that applications such as VoAAL2, with strict Quality of Service (QoS) requirements and asynchronous inputs, can be performed on an NPU.

The VoAAL2 application is most naturally implemented using an asynchronous programming model. We found that the IXP2400 naturally supports such a programming model. Support for asynchronous components that span MEs could be improved by adding support for a distributed lock manager.

The AAL2 application requires a lot of bit- and byte-level data access. The IXP2400 provides all of the necessary facilities to perform these operations while meeting the performance requirements of the application. Finally, the VoAAL2 application requires real-time scheduling to conform to the TM4.1 traffic contract. Although the IXP2400 does not support preemptive scheduling, the software can be tuned to perform the required scheduling in conformance with TM4.1.

REFERENCES

[1] S. Lakshmanamurthy, et. al, "Network Processor Performance Analysis Methodology," *Intel Technology Journal*, Vol. 6 issue 3, August 2002.

[2] ITU-T Recommendation I.363.2, Series I: B-ISDN ATM Adaptation Layer Specification: Type 2 AAL, ITU, Geneva, Switzerland, 1997.

[3] ITU-T Recommendation I.366.2, AAL Type 2 Service Specific Convergence Sublayer for Trunking ITU, Geneva, Switzerland, 1999.

[4] The ATM Forum, "Traffic Management Specification Version 4.1," af-tm-0121.000, April 1999.

[5] F. Baker, "RFC 1812 Requirements for IP Version 4 Routers," *IETF*, June 1995.

AUTHORS' BIOGRAPHIES

Jaroslaw J. Sydir is a network architect in the Silicon Development Group of the Network Processor Division at Intel Corporation. His interests are in the areas of signaling protocols, traffic management, and distributed real-time systems. He received his B.S. in Computer Engineering from Case Western Reserve University in 1988, and an M.S. degree in Systems Engineering from Case Western Reserve University in 1989. He can be reached at jerry.sydir@intel.com

Prashant Chandra is a senior staff network architect in the Software and Systems Engineering group of the Network Processor Division at Intel Corporation. His interests are in the areas of programmable networks, signaling protocols, and traffic management. He received his B.E. degree in Electronics Engineering from Bangalore University in 1991, an M.S. degree in Computer Engineering from West Virginia University in 1994, and a Ph.D. degree in Computer Engineering from Carnegie Mellon University in 2000. He can be reached at prashant.chandra@intel.com

Alok Kumar is a staff software architect in the Software and Systems Engineering group of the Network Processor Division at Intel Corporation. His interests are in the areas of high-speed programmable routers, quality of service, and computer graphics. He received his B.Tech degree in Computer Science from the Indian Institute of Technology, Delhi, in 1999, and his M.S. degree in Computer Science from the University of Texas at Austin in 2001. He can be reached at alok.kumar@intel.com

Sridhar Lakshmanamurthy is a senior staff architect in the Silicon Development Group of the Processor Division at Intel Corporation, focusing on understanding edge/access networking applications, analyzing the performance of Intel's network processor solutions, and defining future enhancements to these solutions. Prior to joining the Network Processor Division, Sridhar focused on platform performance analysis for Intel server chipsets for Xeon™ & Itanium® Product Family (IPF) processors, and system bus specifications for the IPF processors. Sridhar joined Intel in 1993 after receiving an M.S. degree in Computer Engineering from Rice University in Houston, Texas. He can be reached at sridhar.lakshmanamurthy@intel.com

Longsong Lin is the senior staff network architect in the NPG technology office. He was the principal system architecture at AMCC, CTO at Opix Networks, senior research staff and architect at NEC, Japan, scientist at Swiss Federal Institute of Technology, Switzerland, VP of Engineering at Jato International Inc., professor and chairman at National Yunlin University of Science and Technology, Taiwan, visiting professor at the University of Illinois, Urbana Champaign, visiting scholar at Purdue university, and a member of the Taiwan Public TV Organizing Committee. He received his Ph.D. degree and M.S. degree in Electrical Engineering at Purdue University. He is currently involved with several projects at Intel, including modular system platforms and next-generation network processor architecture, as well as business and technology developments in APAC. He can be reached at longsong.lin@intel.com

Muthu Venkatachalam is a network architect in the Software and Systems Engineering group of the Network

Processor Division at Intel Corporation. His interests lie in network processor architecture and programming, QoS algorithms, traffic management, systems modeling and analysis, and keeping pace with the innovations in today's networking industry. He can be reached at muthaiah.venkatachalam@intel.com

Challenges and Methodologies for Implementing High-Performance Network Processors

Ram Bhamidipati, Ahmad Zaidi, Siva Makineni, Kah K. Low,
Robert Chen, Kin-Yip Liu, Jack Dahlgren
Intel Communications Group, Intel Corporation

Index words: network processors, reuse, verification, clock architecture, hierarchical flow, transactor, simulator

ABSTRACT

Moore's law has been the guiding principle for performance and transistor density improvements over the years. While this is true, in the context of network processor development, the challenge is multi-faceted to keep the silicon development on the curve.

This paper describes the challenges for a network processor implementation in each facet of design. The network processor designs adopted the following implementation techniques to manage the design challenges and the Time-to-Market (TTM) schedule:

- Reuse of Intellectual Property (IP).

- Extensive functional validation.

- High-performance clock architecture and design.

- Streamlined hierarchical physical design flow.

- Efficient and cycle-accurate c-model for performance simulation.

A case study of implementation on the IXP2400 design is presented with the above strategies in detail.

The silicon results show that the IXP2400 is a successful design following the stated methods.

INTRODUCTION

Network processors are the emerging class of chips that offer Original Equipment Manufacturers' (OEM) flexibility in creating a wide range of applications. They are targeted to replace expensive and inflexible fixed-function silicon Application-Specific Integrated Circuits (ASIC). The implementation of network processors has to form-fit to the schedule needs of a telecommunication industry moving at the Internet speed. At Intel, we chose the architectural approach of providing a highly integrated and highly programmable solution to customers. This means a lot of functionality is packed into the silicon, thereby increasing its complexity for implementation. In addition to the functionality, the network processor has to operate at the targeted line rates, often running multiple tasks as demanded by the end-user applications. The applications may range from basic L3 forwarding to sophisticated algorithms that are more compute-intensive, as in the case of creating firewalls and intrusion detection services.

Network processors interface to a host of devices to perform their functions: media/switch fabric, PCI for control interface, DRAM for packet storage, Quad Data Rate (QDR) as a fast memory for queues and table lookup, and miscellaneous device support including Universal Asynchronous Receiver/Transmitter (UART) and General Purpose Input/Output (GPIO). A number of parallel microengines work on the data packets executing a specific microcode sequence downloaded into their memories by the control processor. The microengines, the control units, and I/O interfaces are connected via an internal chassis bus, and data movement happens in an efficient manner through arbitration schemes. This is akin to a system-on-chip design.
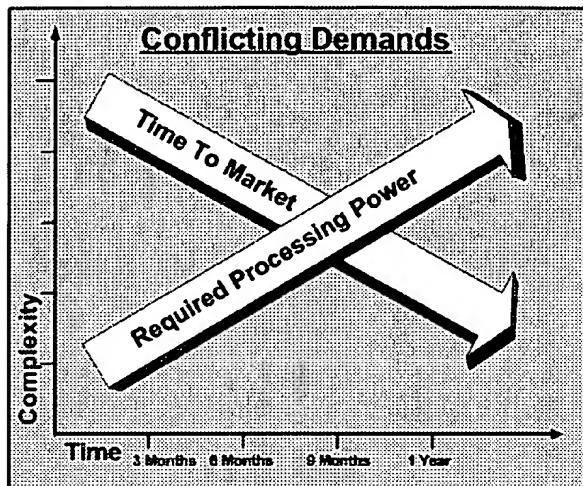
Figure 1: Constantly increasing processing power is required even as the market requires shorter design cycles and time-to-market

A number of challenges for implementation are already evident. As the demand for performance scales up (Figure 1), the number of transistors increases with collateral increases in power and physical design complexity. As an example, the IXP2400 design on p859 packs ~60 million transistors, which is comparable to a high-performance IA32 processor. Further, to meet the performance goals, the critical processing elements such as the microengines and the XScale™ control processor are working at the highest clock rate possible (600MHz in IXP2400). The high-frequency operation requires custom design of Arithmetic and Logic Unit (ALU) and memory elements.

Given the varied usage models in the field, extended temperature range (-40deg.C to 85deg.C) is a Plan Of Record (POR) for network processor implementation at Intel. The power requirements are also stringent, with applications ranging from fully enclosed boxes, as in cellular base stations, to heat-sink solutions on high-performance blades (line cards) in a rack system. These requirements pose a significant challenge for reliability and robustness of the design.

Since network processors have to interface to a lot of I/Os, the result is complexity of package, I/O design, and board design. The requirements on design may be more stringent here in network processors than on a CPU, due to the proprietary nature of designs from OEMs.

Functional verification of a network processor is also a very challenging task due to the fact that it has several interfaces (PCI, DRAM, QDR, slowport, media, switch fabric, etc.) and supports several network protocols. Several on-chip clock domains, both synchronous and asynchronous, make the task even more complicated.

One of the essential deliverables of a network processor design project is a simulator that models the functionality of the network processor with execution-cycle-level accuracy. To external customers and internal software teams, this simulator enables application software development and performance optimization long before the network processor product becomes available in silicon. To the internal design team, this simulator facilitates conducting performance analysis and architecture/microarchitecture studies. The unique nature of network processors poses significant challenges and imposes special requirements on the development of such a simulator. The requirements for the simulator are best illustrated by reviewing the architecture of network processors and the complexity and paradigm of the application development. The simulator must minimize the application development complexity. Furthermore, due to lack of network processor performance benchmarks, the simulator and reference applications must become available at least three quarters before silicon sample date. This schedule enables the potential customers to evaluate the capability of the network processor effectively, and facilitates the committed customers to gain time-to-market advantage by starting application development early. It helps Intel to engage the customers with an architecture before the silicon is available.

In a competitive environment of network processor silicon solutions, Time-to-Market (TTM) becomes a compelling factor for Original Equipment Manufacturers (OEMs) in picking an architecture for product design. For silicon providers, as the performance demand is increasing and the transistor count is thereby increasing, the RTL to GDS2 design cycle times are staying flat. One way to form-fit the complexity of design into the same schedule is by increasing the size of design teams. Studies show that this leads to inefficiencies and increased cost of product development beyond a point.

In response to these challenges, the network processor design teams at Intel have focused on increasing productivity and efficiency in design through reuse, co-development, innovative methodologies, and streamlined tool flows.

The following sections describe IXP2400 as a case study, going into the details of each phase of the network processor design from RTL to GDS2.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

IXP2400 DESIGN: A CASE STUDY

Reuse

The strategy for front-end RTL development was co-development with IXP2800 and reuse of design modules from an Intellectual Property (IP) repository. The design modules for RTL coding were partitioned between IXP2400 and IXP2800 at the beginning of the project execution. The originating project that developed a functional block assigned primary logic owners who are completely responsible for functional correctness. The receiving project assigned secondary owners who are responsible for physical implementation at their end. This co-development was managed well through good interaction at engineering-peer-to-engineering-peer level and between management at the schedule level. Project-specific sub-modules were clearly identified (e.g., reset module). For these sub-modules, a common interface was worked out ahead of time to make it an easily swappable block of code. The least common denomination of memory elements in size and usage was also worked out in this manner. High-performance custom datapath blocks in the microengine were isolated from synthesizable blocks with clear interface partitioning to allow parallel development. It is to be noted here that the IXP2400 and IXP2800 have different process and performance goals. Therefore, the sharing is limited to RTL code.

A number of other functional blocks and sub-blocks have been reused from an IP repository. These include the PCI core, Universal Asynchronous Receiver/Transmitter (UART), XScale™ core (Elkhart) and Double Data Rate (DDR) I/O as shown in Figure 2.

Several IP were harvested for the physical implementation for reuse. For the I/O design, good inventory of IP was available from chipset groups for DDR I/O and a basic I/O buffer design for all the others: Media Switch Fabric (MSF), PCI, and miscellaneous I/O. Quad Data Rate (QDR) I/O was generated by modifying the DDR I/O design. Much of the I/O effort then was focused on integrating the I/O blocks for the chip floorplan and for doing signal integrity checks for the board reference designs.

The analog high-frequency Phase Locked Loop (PLL) design was imported from the chipset group and tuned extensively to IXP2400 requirements.

The basic cells for the SRAM memory elements and the SRAM architecture were reused from a CPU group. This cut down the development time to two quarters.
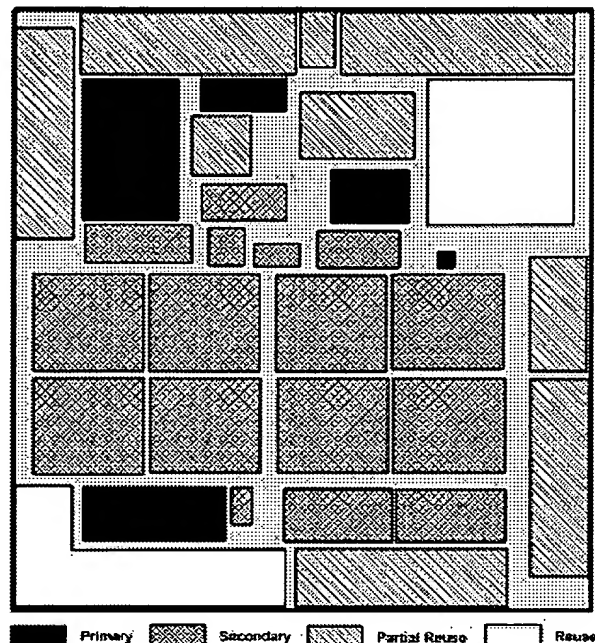

Figure 2: The IXP2400 design is a mix of reuse and co-development

Functional Verification

This section describes some of the methodologies the team adopted to successfully complete verification of the IXP2400 network processor.

Verification of the IXP2400 started with choosing the right tools and verification platform, and defining the verification methodology. The team evaluated several alternatives and chose Cadence's* NCSIM for logic simulator, Verisity's* SPECMAN for test bench automation, X86 Linux platforms for computing servers, Debussy* waveform viewer for debug, and Denali* memory models. A clear methodology was defined and documented with two main goals:

1) The primary goal was to make sure that A0 silicon had adequate functionality that enabled the team to build a system-level environment and run software.

2) The secondary goal was to enable customer sampling on the A-dash stepping.

An extensive upfront methodology was documented with various milestones and exit criteria for each of these milestones. This methodology document defined the rules and guidelines to be followed while developing the verification components to make them extendable, expandable, and readily usable as an IP by another project. IXP2400 and IXP2800 projects used this methodology and

seamlessly shared the verification components. Figure 3   shows the validation view of the Sausalito architecture.
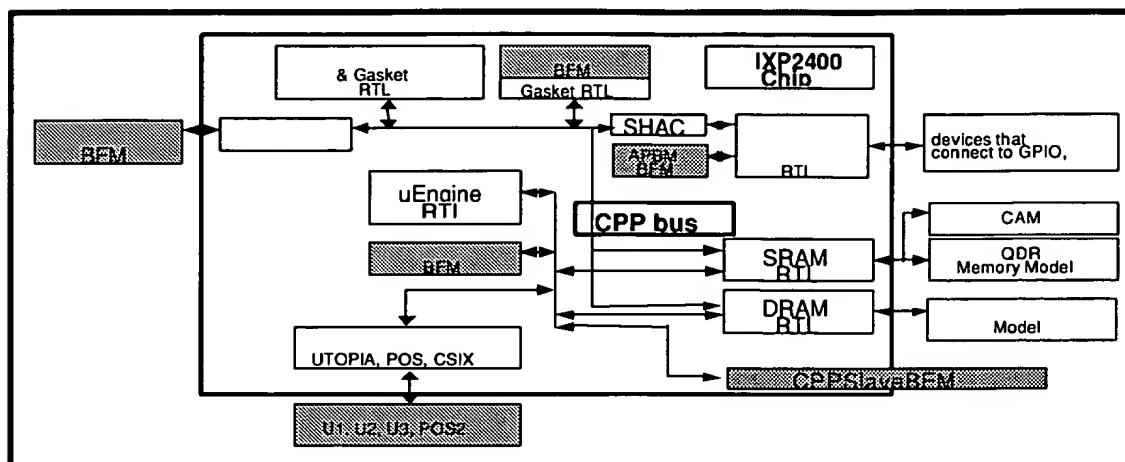

Figure 3: Validation view of IXP2400 architecture

Following are some of the salient features of the IXP2400 verification methodology.

1. Testing design at multiple levels of integration, namely, block level, full-chip, and system levels: System level simulation puts together an IXP2400 full-chip RTL model with the ecosystem surrounding the chip in some real-life applications. The intent of system-level simulation is to make sure that the chip is compatible with ECO system components such as framers, and compliant with industry standard protocols such as UTOPIA and POS-PHY.

2. Monitor-Based Testing (MBT): The real power of SPECMAN lies in its built-in random generator, and this power is used during test plan implementation. Tests are developed in a three-phase approach. In phase 1, simple, directed tests are written to cover the breadth of design. In the second phase, the random power of SPECMAN is unleashed to generate interesting test cases. For each of the test cases in the test plan, monitors are written to make sure that the test case is covered. Hence, all the test cases whose monitors got triggered during this random run are checked off. The coverage report is analyzed to identify the test cases that are not covered. These uncovered test cases are the focus of the third phase, in which directed tests are written to cover them.

3. Random testing: To increase confidence in the model, SPECMAN's random generator was put to use. A concurrent random test environment was

built to generate random transactions on a bus with multiple masters and slaves. This environment is highly configurable to choose specific master(s), slave(s), and type(s) of transactions.

4. Gate-level verification: This was used to weed out initialization deficiencies and synthesis bugs.

5. Error checking that uses three methods: 1) extensive score-boarding techniques were used in packet generators; shadow memory techniques were used for memory data checks during and at the end of each test, 2) for microengine verification, a reference model was written in C and was used to validate the RTL model, and 3) protocol checkers were used to verify the behavior of the design for compliance with certain protocols.

6. Structural and functional coverage monitoring techniques.

7. Automation scripts and web-based regression methodology: these used netbatch tools to balance and distribute jobs across multiple servers.

8. Complete debug: the verification team took a goal to debug the RTL failures in order to determine the root cause. In several cases, the team not only root-caused the failure but also identified the fix. This enabled the team to gain extensive knowledge of the design, which helped during later stages of per-silicon debug and also helping post-silicon debug.

9. Rigorous quality and progress measurement indicators: various indicators were developed to measure the quality and progress. The two kinds of indicators used were 1) trend and 2) snapshot. Trends were useful for determining progress against the plan over several weeks. Snapshots were used to get the status at a single point in time and were used to point out problems in a specific block or area of testing.

IXP2400 CLOCK ARCHITECTURE OVERVIEW

To support operations of various memory interfaces, communication interfaces, and internal computing hardware, IXP2400 has 16 clock domains, not to include various test clocks. The highest clock rate is used by a microengine and the on-die XScale™ core, up to 720MHz, to generate high-computing performance for packet processing. The global communication buses for major internal hardware units operate up to 360MHz. To assist sending and receiving data to external memory devices, including both DRAMs and SRAMs, 1X, 2X and 4X clocks are generated for the memory device controller and IO devices. IXP2400 also has four independent media interface clock regions, operating from 25MHz to 125MHz. The clock rates for communication interfaces and memory interfaces are all programmed through control registers. For boot up and host processor communications, IXP2400 has PCI and slow port interfaces running up to 66MHz and 60MHz, respectively.

To support all these clock domains, IXP2400 has a total of 5 Phase Locked Loops (PLLs). Four of them are for generating independent asynchronous clocks for the four media communication interfaces, and the remaining one is for generating all internal clocks for packet processing and all memory interfaces (Figure 4). With various clock domains, data crossing is done through extensive use of a stepping stone control scheme to ensure safe data crossing, in the presence of higher clock skew between different clock domains. Stepping stone control is also enforced in test mode between clock domains that are normally asynchronous in nature.

The clocking in IXP2400 has numerous features to support testing and debug. A debug counter that counts up to 67 million cycles is incorporated to support a count-down and clock-stopping function, so that the device can stop at a particular cycle and SCAN out of internal states can begin. The clock distribution system supports bypassing of external SYS_CLK, SCAN clocks, and JTAG clock.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

*Other brands and names are the property of their respective owners.

IXP2400 Clock Design

The IXP2400 clock design can be grouped into two parts: the clock generation and the clock distribution.

The clock generation consists of a PLL and a clock divider. The PLL is a leveraged IP that we adapted to fit into the IXP2400 area constraint. The divider was custom built by the IXP2400 team to meet the more stringent requirement of low latency by the IXP2400 chip. Lower latency means lower full-chip clock skew. The reduction of the full-chip clock skew from the divider is estimated at 60ps. To design a fast divider, non-critical paths were carefully designed to still meet their timing yet, more importantly, have minimum impact on or even help speed up critical paths. Logics were combined innovatively and carefully optimized using custom techniques. As a result, the divider clocks are generated after just one latch delay from receiving the source clock.
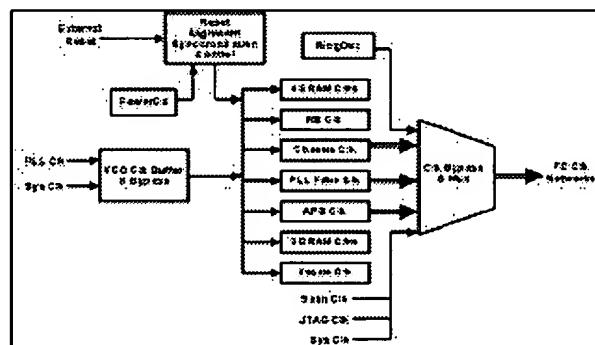

Figure 4: The IXP 2400 clocking scheme

Clock balancing within the divider was made more challenging; given the quest for low latency, more design efforts on balancing were spent after low latency was achieved. Layout was also carefully scrutinized for balancing. A local pre-divide grid was used to reduce RC. Delay elements were added to allow further fine-tuning of the clocks, if necessary. The frequency range of the clocks on IXP2400 is wide; thus a high-ratio divider was designed. An important part of the divider is the error-

correction circuit, which combats noise and ensures clock alignment. The clock dividers are also programmable.

The clock network design challenges were evident from the beginning. The number of clocks in a network processor chip is high compared with general-purpose microprocessors. In addition, the IXP2400 has a large die size; so, inherently, the clock skew would be large if not carefully designed. Low power was another consideration. Tight schedule was another challenge, and quick turn-around time was another goal. These were some of the challenges in designing the IXP2400 full-chip clock network.

For low-power consideration, a balance tree clock network style was selected. It uses a fixed route to stabilize RC and reduce iteration impact to full-chip layout. The large number of clock drivers is grouped into clock station macrocells. Layouts were done with easy programming in mind. Most of the clock stations were designed to drive a fixed load to ease clock tuning. At the chip level, all clocks were routed with shielding. The full-chip clock network RC was extracted and simulated in SPICE. Scripts and automation were developed to quickly tune the clock networks once the block-level clock data are in. Eventually, towards tape-out, the full-chip clock network tuning turn-around time is just one day. At the block level, a pre-grid clock scheme was used to reduce clock skew. RC extraction data were fed back from the block to the full chip for top-level clock tuning. Block-level-clock tuning was automated for smaller blocks and carried out by hand for large blocks and I/Os. SPICE was the primary simulation engine for accuracy of the results. Place and route blocks were tuned by a clock tree synthesis tool for the last two stages of clock networks just before reaching the flops. Clocks lines were plotted and reviewed by the clock owner and the individual block designers.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

IXP2400 HIERARCHICAL DESIGN METHODOLOGY AND FLOW

The primary requirement on the design flow is to enable short Time-to-Market (TTM). Therefore, it is imperative to employ a high level of design automation to increase productivity. We employed the following basic strategies to help achieve the TTM goal:

- o Top-down-driven hierarchical design flow.
- o Cell-based methodology.
- o Streamlined custom circuit-design flow.

We began by performing careful floorplanning at the chip level to obtain an accurate wire model of major/critical signals/busses, block sizes/placements, and location of the block-level pins. A full-chip timing budget was then done to allocate timing constraints to the blocks. The block-level constraints were then passed on to the block designers, who then performed an initial design and provided the feedback to the full-chip designer. By performing upfront planning and getting early bottom-up feedback, we reduced the number of iterations needed to converge on the final design goals.

Secondly, through the use of cell-based methodology with only static CMOS logic, we were able to take advantage of the industry standard Application-Specific Integrated Circuits (ASIC) design tools, namely, logic synthesis and automatic place and route tools, which offer a relatively fast design cycle. These tools were used on most of the blocks, with the exception of the timing critical datapath blocks of the microengines, memory arrays, and IO pads. In addition, we were careful to ensure that flip-flops were used at all block boundaries to minimize inter-block interactions.

The key enabler for achieving high productivity in our custom design flow is a tool suite from MicroMagic (now part of Juniper* Networks). Coupled with the cell-based design methodology, the tool allowed us to specify the relative placements of the cells while composing the schematics of the datapath blocks. The tool automatically generates the block layouts with placed cells. Timing analysis is then done with global routings to obtain the performance of the physical design. The placements of the cells can then be fine-tuned to improve timing where necessary. Once the timing goal is met, an automatic detailed router is used to complete the layout. In this manner, the datapath block layouts were completed with less than one-third the amount of effort compared to full custom-design methodology. Likewise, the memory arrays were constructed through the use of MicroMagic tools, which automatically assemble the array layouts based on a set of high-level commands. Furthermore, the command scripts were parameterized so that arrays of different sizes (within a pre-defined range) could be compiled automatically with minimal efforts.

Last, but not least, the design project was managed using a structured design flow with a series of discrete milestones. The flow diagram in Figure 5 depicts the high-level view of the design flow.

*Other brands and names are the property of their respective owners.
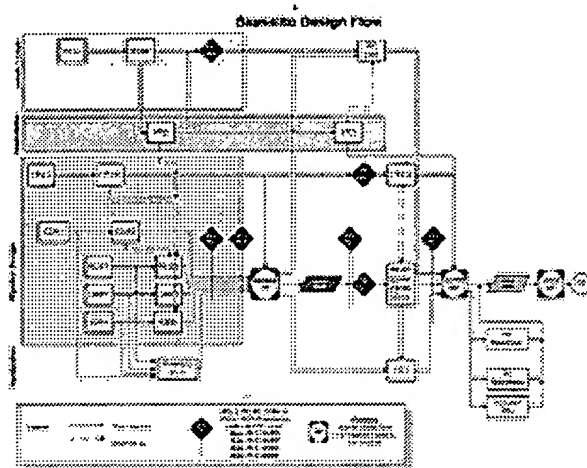
Figure 5: The IXP2400 team used this flow to rapidly converge on the final implementation while maintaining control of the data

IXP2400 Design-for-Test Methodology

IXP2400 is a complex System on a Chip (SOC) design with more than 300 embedded arrays and 88 scan chains encompassing 120K flip-flops spanning across 14 different clocks running and configurable from 33 to 600MHz. The complexity of the system requires a carefully planned Design For Test (DFT) methodology to enable manufacturing and silicon debug. To achieve high test coverage, full-scan design methodology is used throughout the entire chip. Embedded memory arrays are tested using either memory built-in self-tests (MemBIST) or scan-collars. In scan-collared arrays, scan flops are placed on both the input and output stages and are used to control and monitor the arrays. All of the scan-collared arrays are part of the 88 scan chains. Also, boundary scan is implemented on the IO pads to facilitate system-level testing.

To assist in silicon debug, we included a novel scan-debug feature on the chip. This feature allows the chip to run at full speed from reset and stop at a user-defined cycle. The internal state of the machine, i.e., the content of each scanned register, can then be shifted out through the scan chains.

Network Processor Cycle-Accurate Simulator

This section describes the challenges, the requirements, and the successful development strategy and tools that the IXP2400 development team employed.

Architecturally, a network processor consists of clusters of packet processors, co-processors with specific functions, on-chip memory, various kinds of memory and bus interface controllers, and many mechanisms that provide fast communication and signaling among these hardware elements, and a general-purpose CPU, all on a single chip. In the case of IXP2400, the packet processors are called microengines. A microengine is a multi-threaded processor that excels in processing packets at line rate. Each IXP2400 microengine supports up to eight threads of execution. Thread switching is controlled by software and poses zero cycle penalty. IXP2400 contains integrated SRAM and DRAM controllers. Moreover, IXP2400 offers hardware acceleration for managing queues and First In First Out (FIFO) rings, and supports atomic operations for the SRAM and on-chip memory address spaces. Furthermore, IXP2400 provides highly flexible network media and switch fabric interfaces for receiving and transmitting packets.

From the user's perspective, application development for network processors is an exercise of real-time, multi-threaded, and multi-processor programming at the same time. The performance of the application must ensure that the throughput of packet processing exceeds the desired line rate so that packets do not get dropped. In order to create optimized and efficient applications, developers must account for the latency and sequence of all the transactions, as well as the interactions among the various execution threads and hardware units. As a result, the simulator must offer both functional and cycle accuracy. Furthermore, the simulator must monitor a rich list of performance statistics and all the transactions every clock cycle, and must enable the developers to visualize them through an effective Graphical User Interface (GUI).

For the Intel IXP family of network processor products, the simulator is called Transactor, and the GUI tool is called Workbench. Workbench offers the single GUI for code development using assembly or microengine C language, for running simulations to debug and performance-tune applications, and for debugging with the real network processor hardware.

The complexity of network processors, the requirement of 100% cycle accuracy, and the fact that external Transactor releases begin during the early phase of the project all pose significant challenges to the Transactor development team. In addition, the team must achieve excellent development efficiency and quality.

The development strategy that the IXP2400 project employed is based on an internal tool called VMOD. Conceptually, VMOD accepts a logic design at the RTL level and generates the corresponding cycle-accurate C++ model, i.e., Transactor. Moreover, this C++ model supports an API for interfacing Transactor with the workbench. This API relays user commands to Transactor and facilitates communication of model states,

performance statistics, and status of transactions under simulation between Transactor and Workbench.

RTL code presents the functional and cycle-count behavior of a logic design to VMOD. However, RTL code describes only the low-level hardware and does not convey model states at the architecture level. For instance, Transactor users operate with architectural registers, but a register in RTL may be a group of flip-flops that are individually addressed through signal names with long hierarchies. In addition, the RTL code of a logic design does not include performance statistics and does not monitor transactions that execute on top of the hardware that the very RTL code models.

In order to enable Transactor to present architectural states and performance statistics to the users, and to track all the simulated transactions, VMOD accepts C++ code in addition to RTL. This C++ code can read and write individual RTL signals and runs in lock-step with the simulation of the RTL. During simulation, this C++ code collects performance statistics and tracks all the transactions by accessing the relevant RTL signals. Moreover, when the Transactor user wants to access an architectural state, this C++ code translates the mapping of the requested architectural state to the actual collection of RTL signals that make up the architectural state.

Within the IXP2400 design team, the Transactor team owns the development of the Transactor, and the logic design team owns the development of the RTL model. The Transactor team develops the C++ code for all the architectural states, performance statistics, and tracking of transactions. In addition, the Transactor team inputs both the C++ code and the RTL of the logic design into VMOD for Transactor generation. In addition, to ensure excellent quality, the Transactor team builds a thorough regression suite for validating Transactor.

RESULTS

Reuse

Reuse has been a tremendous win overall for the IXP design program. In particular for IXP2400, it cut down the development times for critical elements of design, for example, in I/O and clock design. Co-development of RTL in the front-end has helped IXP2400 and IXP2800 to synergize and develop designs that are completely compatible from the microarchitecture level to the cycle-accurate models on simulators. The sharing of knowledge and resources helped to avoid duplication of effort and kept the design cost low with beneficial affect on the time-to-market schedule.

Pre-Silicon Verification Effort

Sausalito RTL verification was done very efficiently by following a robust methodology. The team completed the pre-silicon verification in approximately nine months after the first RTL model. Sharing verification across IXP2400 and IXP2800 was very beneficial. Quality was never compromised in the verification effort. Results of the work are as follows:

- No functional bug escaped pre-silicon verification after seven weeks of extensive testing on three platforms, namely, the Omaha validation platform, the Angel Island evaluation platform, and the IX/IMS testers.

- Though the two projects IXP2400 and IXP2800 used different simulators and validation platforms, the pre-defined methodology and guidelines allowed then to share the verification components seamlessly.

- Using Linux machines saved several hundreds of thousands of dollars to the division, and it proved that risk-taking pays off.

Clock Architecture

A low clock skew well-balanced clock architecture was achieved through the methodology at the end of the IXP2400 design. The clock tuning also converged rapidly at the end. The silicon probing confirmed the simulated results.

Hierarchical Flow and Methodology

The high-end ASIC design flow that emerged from the IXP2400 design flow enabled rapid physical design convergence at the end of design. The time from RTL closure to the GDS2 database freeze for tapeout was less than one quarter.

Using the hierarchical design methodology and flow described in this paper, the IXP2400 design team was able to implement and complete the chip design in 12 months. This represents a tremendous achievement, considering the complexity and performance level of the chip. The design flow is further validated by a functional first silicon. This proves that a high level of quality is possible with the TTM design flow.

Transactor

With this effective development strategy and the capability of VMOD, the IXP2400 Transactor project has been on schedule since the first external SDK release, which happened more than three quarters before the IXP2400 sample date. Moreover, architects successfully

completed performance analysis by developing reference applications and validating that IXP2400 meets the performance goals during the chip design phase by using Transactor.
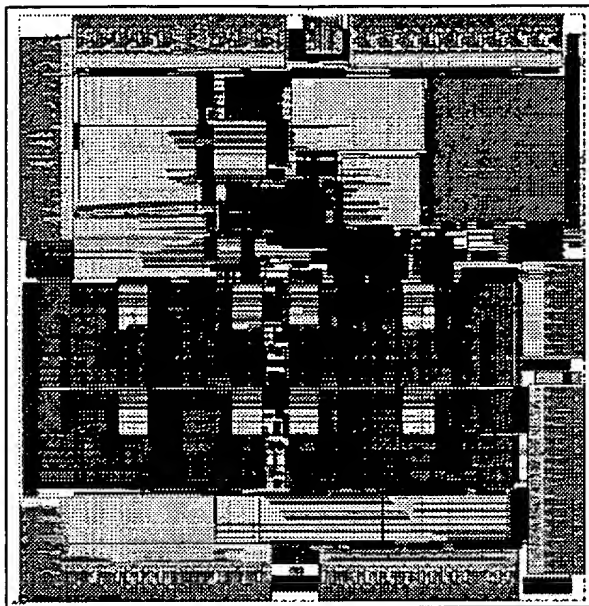

Figure 6: IXP2400 die plot

CONCLUSION

In an emerging and competitive environment of network processor solutions, it is imperative to keep the customers engaged continuously. This interaction starts for the design team with providing an accurate simulator months ahead of time to the actual functional silicon availability. It is also essential to keep the network processor development times on the scale of Moore's law or face extinction.

The network processor design teams have embraced the best-of-class practices to manage the unique design challenges and deliver the products in line with customer expectations. The IXP2400 design (Figure 6) was completed in four quarters from the Implementation Plan Approval (IPA), a goal set at the start of the project. The A0 post-silicon was obtained on schedule. After one quarter of extensive testing on three different platforms, no functional issues have been found. The first customer samples, based on A0 silicon, were shipped out one week ahead of the plan established at IPA.

REFERENCES

[1] C. Narad and L. Huston, "Introduction to Network Processors," Hotchips-12 Presentation, August 2000.

[2] S. Batzer, et. al, "Modeling the Cost Avoidance Potential of a Structured Approach to IP Reuse at Intel," *DTTC* papers, July 2002.

AUTHORS' BIOGRAPHIES

Ram Bhamidipati joined Intel in 1989, after completing his M.S. in Electrical Engineering from N.C.A.&T. State University. He has worked on processor design groups for the development of i486™, Pentium® II, and Itanium® processors. He holds two US patents in design. Currently, he is managing the back-end design of the IXP2400 network processor. His e-mail address is sriram.bhamidipati@intel.com

Ahmad Zaidi joined Intel in 1987, after completing his Master of Electrical Engineering degree from Virginia Tech University. Ahmad is currently Director of Silicon Engineering for NPD-San Jose, focusing on architecture, design, program management, and manufacturing of network processors for the Access and Edge market segments. His prior assignments include engineering management positions on the Itanium Processor, and engineering positions in the i386™, i486, and Pentium® microprocessor projects. Ahmad holds nine US patents in microprocessor design and architecture. His e-mail address is ahmad.zaidi@intel.com

Siva Makineni joined Intel in 1992, after completing his Master of Engineering (E.E.) degree from Worcester Polytechnic Institute in Worcester, Massachusetts. Prior to joining the IXP2400 team as pre-silicon verification manager, Siva held several engineering and management positions in Itanium, Pentium, and 486SL projects. Most recently, he designed floating point arithmetic units on the Itanium processor and managed the SIMD floating point implementation team. Siva holds ten US patents in floating point and integer arithmetic. His technical interests include high-speed floating point architecture and design, computer arithmetic, and developing effective verification strategies. His e-mail address is siva.makineni@intel.com

Kah K. Low is currently the design center manager of Intel's Malaysia Network Development Center, where he leads the development of next-generation network processors. Previously, he was the global design manager in the IXP2400 design project. He joined Intel in 1995 to work on the Itanium design project, where he managed the circuit design automation group. Prior to Intel, Kah K. was with Motorola, Inc. from 1989-1995, where he worked on statistical design, device modeling/characterization, CAD, digital signal processors, and where he served as a project manager in SEMATECH's phase-shifting mask program. He holds three US patents and received his B.S. degree from the University of Massachusetts, and his M.S. and Ph.D. degrees from Carnegie Mellon University, all in Electrical Engineering. His technical interests include network processor design, VLSI design methodology, CAD tools, and communication networks. His e-mail address is kah.k.low@intel.com

Robert Chen received his Ph.D. degree in Electrical Engineering from the University of Notre Dame in 1993. Since then, he has worked in various areas of IC design: library, SRAM, register files, TLB and CAM, low power, clocks, place and route, and logic design. He received "Top Gun Award" from Sun Microsystems, Inc. in 1995, and "IA-64 Processor Division Award" from Intel in 2000 for his work on McKinley power reduction. Currently, Robert is working on the clock design for the next generation of IXP2400. His e-mail is robert.chen@intel.com

Kin-Yip Liu joined Intel in 1990, after completing his Master of Engineering (E.E.), Bachelor of Science (E.E.), and Bachelor of Arts (Economics) degrees from Cornell University. Kin-Yip now co-manages the NPD NPBU Architecture team at San Jose, focusing on network processors for the Access and Edge market segments. His prior assignments include engineering and management positions in the Itanium Product Family architecture and firmware teams and in the 386SL and 486SL microprocessor projects. Kin-Yip holds four US patents in microprocessor architecture. His technical interests include network processing, computer architecture, and simulator development. His e-mail address is kin-yip.liu@intel.com

Jack Dahlgren joined Intel in 1997, after ten years in the architecture and construction management industries. He has provided project control services on several projects including development of Itanium and the Mobile Intel® Pentium® III Processor. His educational background includes Master's degrees in Architecture and Civil Engineering from the University of California at Berkeley. His e-mail address is jack.dahlgren@intel.com

For further information visit:
developer.intel.com/technology/itj/index.htm